Санкт-Петербургский государственный политехнический университет
Институт информационных технологий и управления
Кафедра компьютерных систем и програмных технологий

Диссертация допущена к защите
зав. кафедрой

_____ В.Ф. Мелехин

«___» _____ 2014 г.

ДИССЕРТАЦИЯ
на соискание ученой степени МАГИСТРА

Тема: **Инструментальная среда для анализа программных систем**

Направление: 230100 – Информатика и вычислительная техника
Магистерская программа: 230100.68.15 – Технологии проектирования системного и прикладного программного обеспечения

Выполнил студент гр. 63501/13 _____ А.М. Половцев

Научный руководитель,
к. т. н., доц. _____ В.М. Ицыксон

Консультант по нормоконтролю,
ст. преп. _____ С.А. Нестеров

Санкт-Петербург – 2014

Эта страница специально оставлена пустой.

РЕФЕРАТ

Отчет, 108 стр., 25 рис., 2 табл., 24 ист., 3 прил.

МЕТАМОДЕЛЬ, ИНСТРУМЕНТИРОВАНИЕ, ВИЗУАЛИЗАЦИЯ, КАЧЕСТВО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Магистерская диссертация посвящена разработке средства автоматизации процедур анализа и верификации программных систем. Для решения поставленной задачи был предложен подход на основе языконезависимой метамодели. Для реализации данного подхода был проведен анализ существующих решений и рассмотрены способы проектирования средств метамоделирования.

Для демонстрации возможностей метамодели спроектирована архитектура инструментальной среды для извлечения различных видов моделей и проведения процедур анализа. Разработаны преобразователи для языков Java и C для импортирования моделей анализируемых программных систем. Разработаны алгоритмы визуализации различных свойств анализируемого ПО. Проведено функциональное тестирование всей разработанной системы.

Инструментальная среда может быть использована в некоторых методах повышения качества, таких как аудит, для наблюдения за свойствами разрабатываемой системы. Разработанная метамодель в виде библиотеки может использоваться для разработки процедур анализа и верификации, например, процедур статического анализа.

ABSTRACT

Report, 108 pages, 25 figures, 2 tables, 24 references, 3 appendicies

METAMODEL, INSTRUMENTATION, VISUALISATION,
SOFTWARE QUALITY

This thesis introduces an automation tool for developing software analysis and verification procedures. We propose an approach to automation of the analysis by using a language-independent metamodel. In order to implement this approach, we analyse existing tools and methods of designing the architecture of metamodeling systems.

To demonstrate the capabilities of the metamodel, we have desinged an instrumentation system for exctracting models and running various analysis procedures. We have developed Java and C programming language translators for importing analysed software systems into the tool and multiple software properties visualization algorithms. After everything is done, functional testing of the whole system is performed.

Implemented instumentation tool can be used in such software quality assurance methods as software audit review to monitor software properties. The metamodel library is useful for designing new analysis and verification procedures.

СОДЕРЖАНИЕ

СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

ANTLR	ANother Tool for Language Recognition, инструмент для генерации лексических и синтаксических анализаторов
API	Application programming interface, интерфейс прикладного программирования
AST	Abstrac Syntax Tree, абстрактное синтаксическое дерево
CFG	Control Flow Graph, граф потока управления
CMOF	Complete MOF, основная структурная составляющая MOF
DDG	Data Dependency Graph, граф зависимостей по данным
DTD	Document Type Definition, описание схемы документа для языка разметки
eCST	enriched Concrete Syntax Tree, модель программы, используемая средством SMIILE
EMOF	Essential MOF, базовая структурная составляющая MOF
FAMIX	Семейство метамоделей, используемых средством Moose
JavaCC	Java Compiler Compiler, генератор синтаксических анализаторов
JAXB	Java Architecture for XML Binding, фреймворк для XML-сериализации для языка Java
Jparsec	генератор синтаксических и лексических анализаторов
LLVM	Low Level Virtual Machine, фреймворк для создания компиляторов
MDA	Model-Driven Architecture, подход к разработке программных систем
MDE	Model-Driven Engeneering, разработка, управляемая моделями
MOF	Meta-Object Facility, стандарт для разработки, управляемой моделями

Moose	Платформа для анализа программ
MSE	язык разметки, используемый средством Moose
OMG	Object Management Group, консорциум, занимающийся разработкой и продвижением объекто-ориентированных технологий и стандартов
POM	Project Object Model, способ описания проекта в система автоматизированной сборки Maven
SDL	Specification and Description Language, язык для описания моделей программных систем
Simple	фреймворк для XML-сериализации для языка Java
SMIILE	Software Metrics Independent of Input LanguagE, средство для анализа программ
SSA	Static Single Assignment, модель программы: однократное статическое присваивание
ULF-Ware	Unified Language Family softWare, фреймворк для генерации кода из программных моделей
UML	Unified Modeling Language, унифицированный язык моделирования
XMI	XML Metadata Interchange, формат для сериализации метамоделей в формат XML
XML	eXtensible Markup Language, расширяемый язык разметки
XMLBeans	фреймворк для XML-сериализации для языка Java
XStream	фреймворк для XML-сериализации для языка Java
ПО	Программное обеспечение
РБНФ	Расширенная Форма Бэкуса-Наура

ВВЕДЕНИЕ

В данной работе рассматривается подход к автоматизации процесса проведения анализа и верификации программных систем с целью повышения характеристик качества.

С развитием вычислительных систем и ростом в них доли программной составляющей, сложность разрабатываемых программ постоянно возрастает. Также, вследствие большой конкуренции на рынке программного обеспечения, постоянно снижаются сроки разработки новых версий ПО. Эти факторы неизбежно ведут к снижению качества выпускаемых продуктов.

Падение уровня качества является проблемой, особенно если программное обеспечение задействовано в критически важных сферах человеческой деятельности, например, медицине и космонавтике, так как наличие в них ошибок ведет к большому материальному ущербу и даже человеческим жертвам. Поэтому задача повышения качества является одной из самых актуальных в сфере информационных технологий.

Одними из способов повышения качества программ являются аудит, статический анализ и формальные методы, которые часто реализуются в виде инструментальных средств. При разработке данных средств часто решаются похожие задачи, такие как:

- Построение моделей программы, например, абстрактного синтаксического дерева, графа потока управления, графа программных зависимостей и т.д. (модели программ рассмотрены в подразделе 1.4)
- Построение различных метрик программного кода
- Реинжиниринг программного обеспечения (оптимизация, рефакторинг)
- Визуализация свойств программной системы

Обычно эти задачи решаются вручную каждый раз при создании анализаторов или проведения верификации программы. В данной работе предлагается способ автоматизации решения этих задач на основе использования представления программы, не зависящего от языка написания ее исходного кода.

В разделе 1 приведены способы повышения качества, используемые в них модели ПО, предложен подход к автоматизации решения задач повышения качества, рассмотрены существующие решения. Раздел 2 посвящен формулированию требований к инструментальной среде и постановке задач. В разделе 3 рассматривается архитектура инструментальной среды и всех ее составляющих. Разработка системы приведена в разделе 4. Раздел 5 посвящен тестированию разработанной системы.

1. АНАЛИЗ ПОДХОДОВ К ПОСТРОЕНИЮ СРЕДСТВ ПОВЫШЕНИЯ КАЧЕСТВА

В данном разделе рассматриваются методы повышения качества и их классификация, ставится проблема их автоматизации. Для решения поставленной проблемы рассматривается подход на основе метамоделей и производится анализ существующих решений.

1.1. Понятие качества программного обеспечения

Качество ПО - достаточно абстрактное понятие. Многие понимают под ним, например, ПО, которое не содержит ошибок. Однако это лишь одна из характеристик качества. В стандарте ISO 8402:1994 “Quality management and quality assurance” [1] дается следующее определение качества:

Качество программного обеспечения - это совокупность характеристик ПО, относящихся к его способности удовлетворять установленные и предполагаемые потребности.

Характеристики качества программного обеспечения - набор свойств программной продукции, по которым ее качество описывается и оценивается.

На данный момент наиболее распространена и используется многоуровневая модель качества программного обеспечения, представленная в наборе стандартов ISO 9126. На верхнем уровне выделено 6 основных характеристик качества ПО, каждую из которых определяют набором атрибутов, имеющих соответствующие метрики для последующей оценки (рис. 1.1).

Рисунок 1.1. Модель качества ПО

1.2. Методы повышения качества

В соответствии с определением, данном в подразделе 1.1, методы повышения качества - это набор подходов по улучшению характеристик качества разрабатываемого ПО. Существует две группы подходов по обеспечению качества программного обеспечения [2]:

1. Подходы, основанные на синтезе ПО
2. Подходы, основанные на анализе уже созданного ПО

Подходы, основанные на синтезе ПО, используют различные формализации во время проектирования системы, таким образом позволяя избежать ошибок на более поздних этапах разработки.

Данные формализации включают в себя:

- формальные спецификации
- формальные и неформальные описания различных аспектов программной системы
- архитектурные шаблоны и стили
- паттерны проектирования
- контрактное программирование
- аннотирование программ
- верификация моделей программ с использованием частичных спецификаций
- использование моделей предметной области для автоматизации тестирования программ

Подходы, основанные на анализе уже созданного ПО, используются для повышения качества уже созданного ПО, что позволяет улучшить огромное количество уже разработанных программных систем, имеющих проблемы с уровнем качества. Данная группа подходов оперирует функциональными и нефункциональными требованиями к разработанной системе для доказательства соответствия или приведения контрпримеров, показывающих несоответствие поставленным требованиям.

1.3. Классификация методов обеспечения качества

Обычно выделяют следующие базовые классификации методов обеспечения качества [2]:

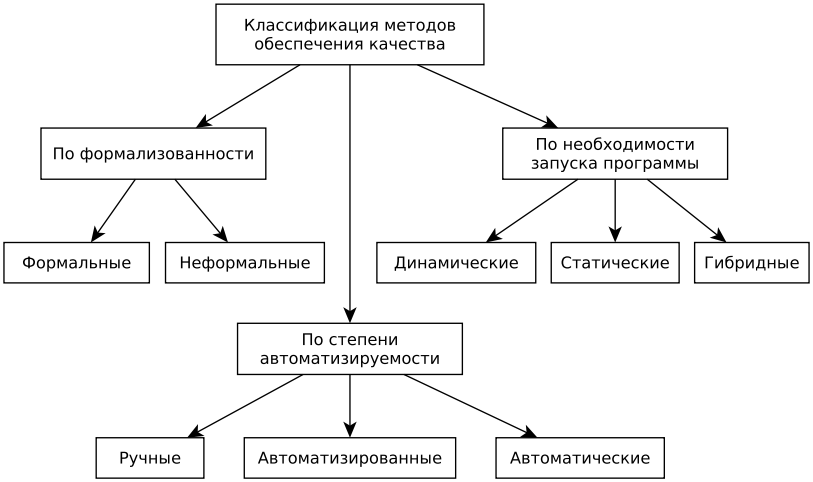

Рисунок 1.2. Схема используемой классификации методов обеспечения качества

По *формализованности* процедур обеспечения качества методы подразделяются на следующие категории:

Формальные методы позволяют создавать формальные функциональные спецификации и модели архитектуры систем, а также осуществлять их преобразование в программы с последующей верификацией [3]. Корректность полученных результатов гарантируется математическим аппаратом. К таким методам относятся, например, дедуктивная верификация, проверка моделей и абстрактная интерпретация.

Эти методы можно применить только к тем свойствам, которые можно выразить в рамках некоторой математической модели. Построение этой модели не автоматизируется, а провести анализ таких моделей может лишь специалист. Однако сама проверка свойств может быть автоматизирована и позволяет находить даже самые сложные ошибки.

В отличие от формальных методов **неформальные методы** позволяют находить ошибки, используя различные артефакты жизненного цикла системы. К их недостаткам можно отнести невозможность

автоматизации [4]. Примером неформального метода является аудит.

По *степени автоматизируемости* методы подразделяются на следующие группы:

Ручные методы - группа методов, не подлежащая автоматизации. К этому классу методов обычно относятся все неформальные методы (аудит, парное программирование).

Автоматизированные методы предполагают частичную автоматизацию процесса повышения качества. Данные методы предполагают проведение ручных действий (например, по подготовке модели в методе проверки моделей) и запуск автоматизированных процедур (над подготовленными моделями).

Автоматические методы не требуют вмешательства специалистов во время своего выполнения, что позволяет легко интегрировать их в жизненный цикл разработки ПО. Примером таких методов является модульное регрессионное тестирование.

По *необходимости запуска* анализируемой программы выделяют следующие классы методов:

Статические методы позволяют находить ошибки и недочеты в исходном коде программ или каких-либо иных артефактов (например, формальные спецификации и аннотации). От остальных методов их отделяет то, что статические методы использует только исходные тексты программы, что позволяет обнаруживать ошибки на стадии написания кода. Таким образом, при анализе отсутствует спецификация программы - описание того, что она делает. Это уменьшает множество обнаруживаемых ошибок, но позволяет полностью автоматизировать процесс анализа. Примером статических методов является статический анализ.

Динамические методы используются для анализа и оценки свойств программной системы по результатам ее реальной работы. Одними из таких методов являются тестирование и анализ трасс исполнения.

Для применения данных методов необходимо иметь работающую систему (или ее прототип), поэтому их нельзя использовать на ранних стадиях разработки. К тому же данные методы позволяют найти только те ошибки в ПО, которые проявляются в его работе.

Гибридные методы объединяют в себе элементы некоторых способов повышения качества, описанных выше. Примерами таких методов являются тестирование на основе моделей (model driven testing) [5] и мониторинг формальных свойств (runtime verification) [6].

Цель таких методов - объединить преимущества уже используемых подходов.

1.4. Модели программных систем

Одной из важнейших составляющих анализа программных систем является построение модели. Без нее анализатор будет вынужден непосредственно оперировать с исходным кодом, что влечет за собой усложнение процедур анализа и самого анализатора в целом.

В зависимости от способа построения и назначения модели, они могут различаться по структуре и сложности и обладать различными свойствами. Существуют следующие виды моделей [2]:

- Структурные модели
- Поведенческие модели
- Гибридные модели

Структурные модели во основном используют информацию о синтаксической структуре анализируемой программы, в то время как поведенческие - информацию о динамической семантике. Гибридные модели используют оба этих подхода.

Структурные модели

1. Синтаксическое дерево

   Синтаксическое дерево является результатом разбора программы в соответствии с формальной грамматикой языка программирования. Вершины этого дерева соответствуют нетерминальным символам грамматики, а листья - терминальным.

2. Абстрактное синтаксическое дерево

   Данная модель получается из обычного синтаксического дерева путем удаления нетерминальных вершин с одним потомком и замены части терминальных вершин их семантическими атрибутами.

Поведенческие модели

1. Граф потока управления

   Граф потока управления представляет потоки управления программы в виде ориентированного графа. Вершинами графа являются операторы программы, а дуги отображают возможный ход исполнения программы и связывают между собой операторы, выполняемые друг за другом.

2. Граф зависимостей по данным

   Граф зависимостей по данным отображает связь между конструкциями программы, зависимыми по используемым данным. Дуги графа соединяют узлы, формирующие данные, и узлы, использующие эти данные.

3. Граф программных зависимостей

   Данная модель объединяет в себе особенности графа потока управления и графа зависимости по данным. В графе программных зависимостей присутствуют дуги двух типов: информационные дуги отображают зависимости по данным, а дуги управления соединяют последовательно выполняемые конструкции.

4. Представление в виде SSA

   Однократное статическое присваивание (static single assignment) - промежуточное представление программы, которое обладает следующими свойствами:

   - Всем переменным значение может присваиваться только один раз.
   - Вводится специальный оператор ϕ-функция, который объединяет разные версии локальных переменных.
   - Все операторы программы представляются в трехоперандной форме.

Гибридные модели

1. Абстрактный семантический граф

   Данная модель является расширением абстрактного синтаксического дерева путем добавления дуг, отражающих некоторые

семантически свойства программы, например, такие дуги могут связывать определение и использование переменной или определение функции и ее вызов.

1.5. Метамодели

Обобщенное решение задач анализа и верификации основывается на использовании *метамоделей* [7]. В подразделе 1.4 были приведены возможные модели, которые могут быть извлечены из описания системы на каком-либо языке программирования и отображающие различные ее свойства. Аналогичным образом, как модели определяют язык описания системы, метамодели определяют язык описания моделей.

Для решения задач проведения различного вида анализа необходимо разработать такую метамодель, которая бы описывала необходимый набор моделей. Наиболее подходящей метамоделью является описание на уровне сущностей анализируемой системы (объектов или функций) и взаимодействия между ними (вызов функции, обращение к полю объекта). Детализация взаимодействия между сущностями при этом должна соответствовать поставленным задачам, т.е. существует проблема поиска компромисса между степенью детализации метамодели (количеством информации, которое можно из нее извлечь) и уровнем абстракции (независимостью от способа описания анализируемой системы). С увеличением степени детализации повышается мощность модели, но тем самым увеличивается ее сложность, так как необходимо учитывать специфические для каждого языка возможности описания взаимодействия объектов программной системы. С ростом же уровня абстракции уменьшается класс задач, для которых применима данная метамодель.

Таким образом, приведенные ранее проблемы можно решить, разработав метамодель, которая бы описывала набор моделей, хранящий необходимое количество информации для проведения анализа, но обладающей достаточной простотой для описания широкого круга программных систем.

1.6. Анализ существующих решений

На данный момент существует лишь небольшое количество инструментов, использующих метамоделирование. Необходимо сравнить

их и выявить их пригодность к решению поставленных задач.

1.6.1. Критерии сравнения

Для сравнения существующих средств введем следующие критерии оценки:

Таблица 1.1. Критерии сравнения средств метамоделирования

№	Критерий	Описание
1	Извлечение моделей программ	Возможность получения различных моделей (AST, CFG) из анализируемой программы
2	Подсчет метрик	Возможность расчета метрик
3	Визуализации	Наличие различных визуализаций свойств системы
4	Свойства метамодели	Свойства используемой метамодели - независимость от языка, простота использования
5	API	Язык программирования, на котором написан API системы
6	Открытость	Возможность бесплатного использования и наличие открытых исходных кодов

1.6.2. Фреймворк Moose

Moose является платформой для анализа программ и поддерживает большое количество различных видов анализа [8]:

1. Построение и визуализация метрик.
2. Обнаружение клонов.
3. Построение графа зависимостей между пакетами.
4. Вывод словаря, используемого в проекте.

5. Поддержка браузеров исходного кода.

Moose использует целое семейство метамоделей под названием FAMIX. Данное семейство обладает довольно сложной структурой, упрощенный вид которой приведен на рис. 1.3:

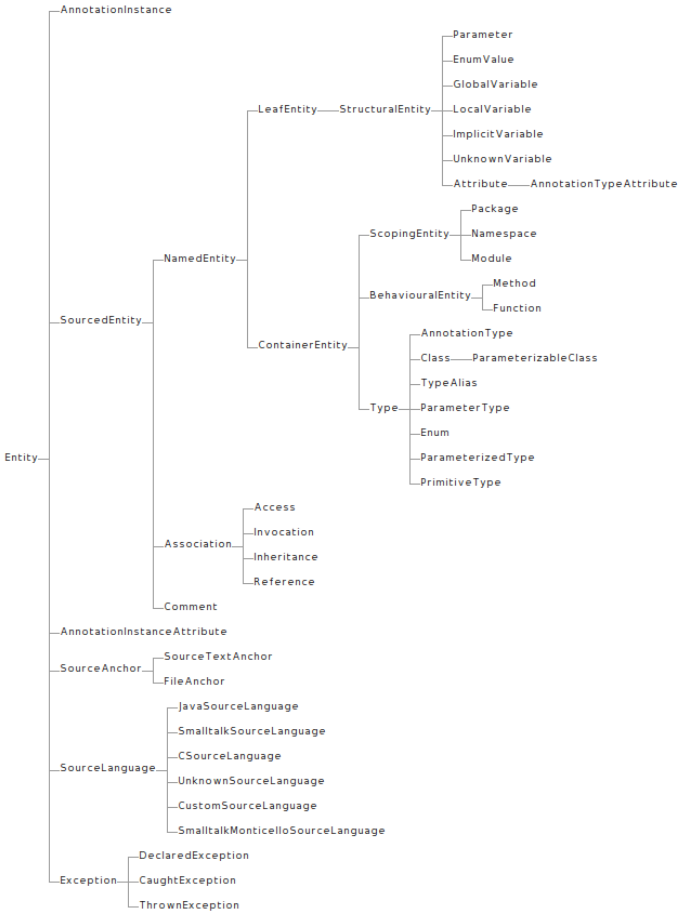

Рисунок 1.3. Структура метамоделей семейства FAMIX

Анализ программы происходит следующим образом:

1. Импортирование входных данных. Импортирование может происходить как при помощи встроенных (Moose поддерживает Smalltalk, XML и MSE), так и сторонних средств.

2. После импортирования данные приводятся к одной из метамоделей семейства FAMIX.

3. Применение заданных алгоритмов анализа.

Архитектура средства приведена на рис 1.4

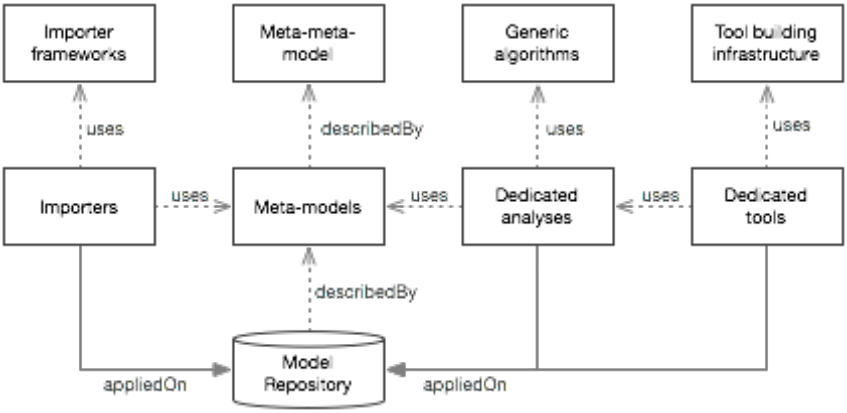

Рисунок 1.4. Архитектура фреймворка Moose

К недостаткам данной среды можно отнести то, что Moose нацелен в первую очередь на задачи реинжиниринга и обладает слишком избыточной и громоздкой метамоделью для разработки на ее основе алгоритмов статического анализа и верификации.

1.6.3. Средство SMIILE

SMIILE - среда, предназначенная для вычисления метрик анализируемой системы [9].

В качестве метамодели SMIILE использует представление в виде eCST (enriched Concrete Syntax Tree), которое представляет собой дерево разбора программы с добавлением универсальных узлов, что

позволяет сделать eCST независимым от языка программирования, на котором написана анализируемая система.

На рис 1.5 изображена архитектура данного средства:

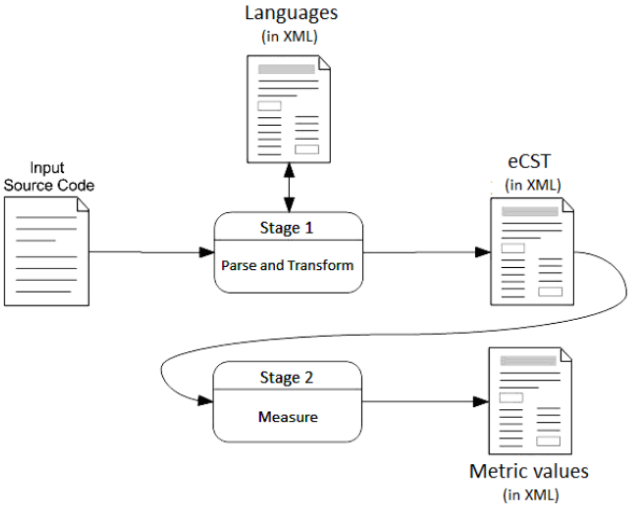

Рисунок 1.5. Архитектура SMIILE

Анализ программы происходит в две фазы:

1. Фаза 1
    а. Определение языка программирования, на котором написана анализируемая программа.
    б. Вызов парсера этого языка для построения CST и преобразование в eCST.
    в. Вывод результата в формате XML.
2. Фаза 2
    а. Считывание eCST из файла.
    б. Подсчет метрик.
    в. Сохранение результата в формате XML.

Средство SMILE на данный момент не разрабатывается и, помимо подсчета метрик, не предоставляет никаких других возможностей.

1.6.4. Фреймворк LLVM

LLVM - фреймворк для анализа и трансформации программ. Данный фреймворк предоставляет информацию для трансформаций компилятору во время компиляции, линковки и исполнения [10].

LLVM использует промежуточное представление, в основе которого лежит представление в виде SSA. Промежуточное представление является набором RISC-подобных команд и содержит дополнительную информацию более высокого уровня, например информацию о типах и графе потока управления.

Данный фреймворк обладает следующими особенностями:

- Сохранение информации о программе даже во время исполнения и между запусками.
- Предоставление информации пользователю для профилирования и оптимизации.
- Промежуточное представление не зависит от языка программирования.
- Возможность оптимизации всей системы в целом (после этапа линковки).

Архитектура LLVM приведена на рис 1.6:

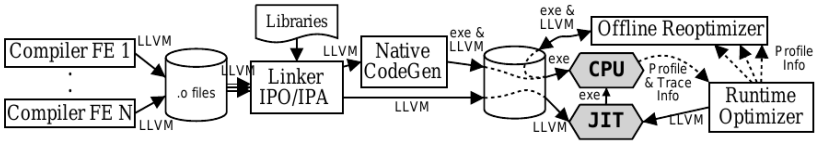

Рисунок 1.6. Архитектура LLVM

Front-end компиляторы транслируют исходную программу в промежуточное представление LLVM, которое затем компонуется LLVM-линковщиком. На этой стадии может проводиться межпроцедурный

анализ. Получившийся код затем транслируется в машинный код для целевой платформы.

Однако, промежуточное представление, используемое в LLVM, обладает недостаточной полнотой описания, необходимой для подсчета метрик и визуализации свойств программной системы.

1.6.5. Фреймворк ULF-Ware

ULF-Ware является средством для генерации кода из модели, созданной при помощи языка SDL (Specification and Description Language) [7]. Данное средство использует метамодель под названием CeeJay, которая предназначена для описания систем на языках Java и C++.

Архитектура ULF-Ware приведена на рис 1.7.

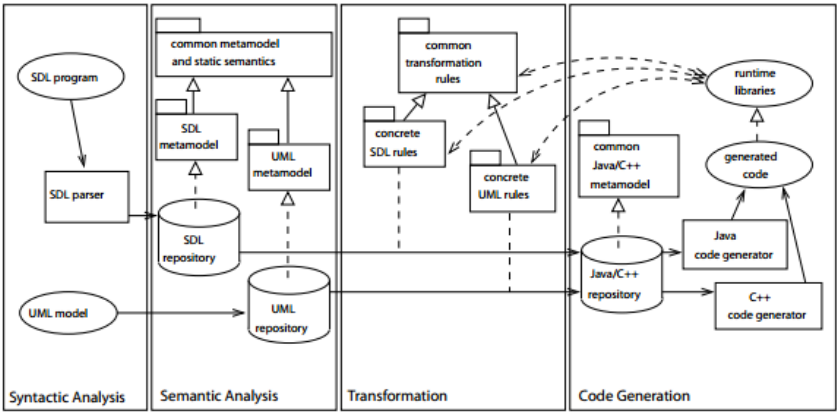

Рисунок 1.7. Архитектура ULF-Ware

Генерация кода происходит следующим образом: компилятор SDL создает модель программы, соответствующую метамодели SDL. После этого происходит трансформация исходной модели в экземпляр метамодели CeeJay, который обладает всей необходимой информацией для генерации кода на языке Java или C++.

Ограничением данного средства и метамодели CeeJay является тот факт, что, в силу специфики задачи, данная метамодель может

применяться только для языков Java и C++.

1.6.6. Результаты анализа

Результаты сравнения существующих средств на основе критериев из подраздела 1.6.1 приведены в таблице 1.2.

1.7. Вывод

В ходе данного раздела были рассмотрены методы повышения качества и модели программ, используемые в этих методах, а также была описаны проблемы автоматизации их реализации. Для решения данных проблем был предложен подход на основе метамодели, абстрагированной от языка описания программ, и проведен анализ существующих решений.

На основе проведенного анализа было выявлено, что наиболее подходящим средством (из рассмотренных) для решения поставленных проблем является среда Moose. Однако, метамодель, используемая Moose отличается высокой сложностью взаимодействия и работы с ней, так как она предназначена для решения слишком широкого круга задач. К тому же данное средство ориентировано в основном на визуализацию свойств, мало связанных с задачами статического анализа и верификации.

Поэтому, для решения задач построения моделей программ и подсчета метрик было принято решение о разработке языконезависимой метамодели и инструментальной среды, позволяющей извлекать эти модели и визуализировать свойства анализируемого ПО.

Таблица 1.2. Результаты сравнения существующих решений

Название	Модели	Метрики	Визуализации	Свойства	API	Лицензия
Moose	-	Метрики размера (количество методов, количество атрибутов, количество свойств)	Визуализация метрик, наличия клонов, зависимостей между пакетами и т.д.	Не зависит от языка, высокий уровень сложности использования	Smalltalk	BSD и MIT
SMIILE	-	Метрики Холстеда, ОО-метрики, цикломатическая сложность и т.д.	-	Не зависит от языка	- (средство не доступно)	-
LLVM	SSA, CFG	-	-	Не зависит от языка, имеет модель в виде SSA	C++	NCSA Open Source License
ULF-Ware	-	-	-	Только языки Java и C++	- (средство не доступно)	-

2. ПОСТАНОВКА ЗАДАЧИ

Как было сказано в разделе 1.4, инструменты для проведения статического анализа и верификации используют различные модели программ для облегчения процедур анализа. Однако данные модели зависят от языка, на котором написана система, а, следовательно, при таком подходе невозможно обобщить разработанные алгоритмы.

К тому же проблема зависимости процедур анализа программного обеспечения от исходного текста остро стоит не только при проведении верификации, но и при решении задач реинжиниринга и оптимизации, а именно:

- Построение метрик
- Визуализация свойств системы
- Поиск клонов
- Анализ истории проекта

2.1. Формулирование требований к инструментальной среде

Таким образом, исходя из описанных проблем, ставится задача разработки среды, которая бы позволила абстрагировать алгоритмы анализа и реинжиниринга от языка описания системы. Это позволит применять разработанные средства для гораздо более широкого круга систем, автоматизировав процесс извлечения модели.

Ядром всей системы и средством абстракции является метамодель, которая должна обладать следующими свойствами:

1. Независимость от языка описания системы - метамодель должна поддерживать несколько парадигм программирования (как минимум, структурную и объектно-ориентированную) и обладать достаточной полнотой для описания специфических конструкций конкретного языка.

2. Расширяемость - возможность добавлять новые сущности по мере необходимости.

3. Простота в использовании - метамодель должна предоставлять удобный API для облегчения обхода структуры модели и реализации различных процедур анализа ПО. Исходя из возможного изменения структуры метамодели API должен предусматривать возможность работы с добавленными узлами.

4. Полнота - метамодель должна содержать достаточное количество информации для извлечения различных моделей, описанных в подразделе 1.4, а также других свойств анализируемой системы (например, метрик).

Для демонстрации возможностей метамодели необходимо разработать инструментальную среду, позволяющую выполнять следующие действия:

1. Визуализация извлеченных моделей

2. Подсчет метрик

3. Визуализация дополнительных свойств системы (например, диаграммы классов для объектно-ориентированной системы)

4. Импортирование систем на языках Java и C (являющимися наиболее популярными языками программирования и реализующие две разные парадигмы)

2.2. Решаемые задачи

Как видно из обзора в подразделе 1.6, ни одно из рассмотренных средств не отвечает поставленным требованиям, поэтому было принято решение о написании собственной инструментальной среды. При этом предполагается решить следующие задачи:

1. Проектирование структуры метамодели, отвечающей всем поставленным требованиям

2. Проектирование графической инструментальной среды, поддерживающей все необходимые виды визуализации

3. Реализация и тестирование программной системы

В результате успешного выполнения данных задач, необходимо получить метамодель, позволяющую описывать программные системы, абстрагируясь от использованного языка программирования, а также инструментальную среду для визуализации ее различных свойств.

2.3. Вывод

В данном разделе были сформулированы требования к разрабатываемой программной системе, а также поставлены задачи, которые необходимо решить для достижения цели диссертации - разработки способа автоматизации процедур анализа и визуализации свойств программных систем.

3. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ИНСТРУМЕНТАЛЬНОЙ СРЕДЫ

При проектировании архитектуры инструментальной среды необходимо учитывать следующие особенности:

- Часть программы, зависящую от языка программирования, на котором написано анализируемое ПО, необходимо вынести в отдельную составляющую для упрощения добавления новых преобразователей из описания программных систем в модель программ, тем самым увеличивая множество доступных систем для анализа.
- Необходимо отделить метамодель от процедур анализа. Таким образом можно расширять набор процедур анализа и визуализации, не изменяя структуру метамодели.
- Крайне сложно найти компромисс между уровнем абстракции и степенью детализации метамодели, поэтому необходимо предусмотреть возможность расширения путем добавления новых узлов в существующую метамодель.

3.1. Структура программной системы

На основе анализа поставленных требований, существующих решений и перечисленных замечаний была предложена следующая архитектура программной системы:

Как видно из рис. 3.1 система состоит из трех частей:

1. Центральная часть системы - *метамодель*. Над ней производятся все операции по проведению анализа и графического отображения свойств анализируемой системы.
2. *Преобразователи* отвечают за импортирование исходного кода анализируемой системы и его преобразование в промежуточное представление (сериализованную метамодель). Преобразователи являются единственной частью системы, зависящей от языка программирования, на котором написана анализируемая система. Однако, так как инструментальная среда оперирует только

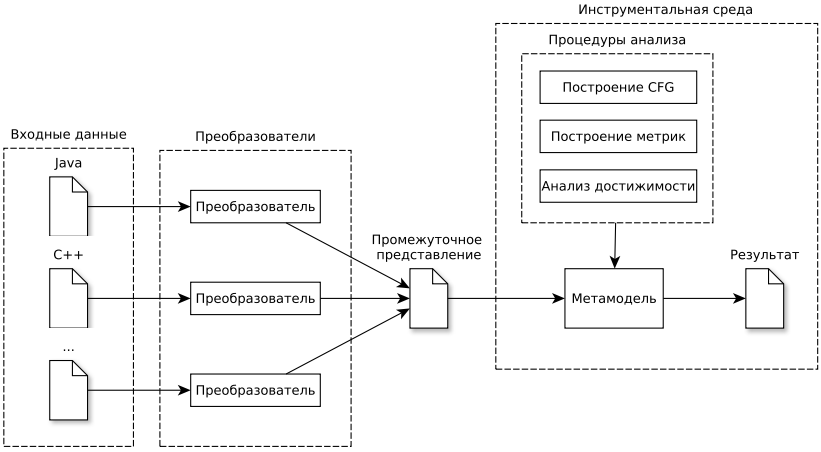

Рисунок 3.1. Архитектура программной системы

с промежуточным представлением, преобразователи могут поставляться сторонними разработчиками, тем самым избавляя программиста от необходимости поддерживать все возможные языки.

3. *Процедуры анализа* вместе с *метамоделью* являются ключевыми составляющими инструментальной среды. Инструментальная среда содержит графический интерфейс пользователя, на котором в том или ином виде отображается результат проведенного анализа.

3.2. Проектирование метамодели

Существует два подхода к разработке метамодели:

1. Разработка архитектуры “с нуля”

2. Использование стандартных средств описание метамоделей, т.н. мета-метамоделей.

Каждый из двух подходов обладает своими достоинствами и недостатками, а именно:

1. Собственная метамодель позволит сильно упростить ее структуру, специализировав ее под нужды инструментальной среды, что увеличит производительность разрабатываемых алгоритмов над метамоделью и облегчит ее использование. Однако данный подход обладает очень высокими рисками, так как данная составляющая является ключевой в разрабатываемой среде, и ошибки в ее проектировании могут привести к провалу проекта.

2. Использование стандартных средств менее подвержено рискам, но, в силу универсальности данного подхода, получившаяся метамодель может быть слишком громоздкой в использовании.

В результате анализа было принято решение об использовании стандартной архитектуры, но с незначительными изменениями, чтобы скомбинировать достоинства обоих подходов.

3.2.1. Стандарт MOF

Рассмотрим подробнее существующий стандарт разработки метамоделей:

В 1997 году группой OMG был создан унифицированный язык моделирования (UML). UML позволяет описывать различные компоненты и артефакты системы, тем самым упрощая процесс разработки [11].

Для описания конструкций языка UML был разработан фреймворк MOF (Meta Object Facility) [12]. В дальнейшем обе эти концепции вошли в подход, называемый Model Driven Architecture (MDA) [13]. Данный подход к разработке ПО вводит дополнительный уровень абстракции, позволяющий описывать структуру и поведение разрабатываемой системы, не завися от нижележащей используемой технологии.

Таким образом, MOF является мета-метамоделью для описания метамоделей (например, метамодели UML). Аналогично расширенной форме Бэкуса-Наура (РБНФ), которая задает грамматику языка программирования, MOF позволяет задавать структуру и абстрактный синтаксис метамодели.

На данный момент последней версией стандарта является версия 2.4.2, выпущенная в 2014 году [12].

3.2.2. Иерархия моделей MOF

Архитектура MOF определена в контексте иерархии моделей, на верхнем уровне которой и находится MOF. Данная иерархия выглядит следующим образом:

M3 - слой мета-метамоделей

M2 - слой метамоделей

M1 - слой моделей

M0 - слой времени исполнения

Слои M3 и M2 также называются *слоями спецификации языков* [14]. Каждый слой является уровнем абстракции - чем ниже слой, тем конкретнее описывается система.

На слое M3 находится только одна мета-метамодель - MOF, задача которой - описание метамоделей. На уровне M2 располагается множество метамоделей, которые являются экземплярами MOF. Слой моделей содержит пользовательское описание системы. Слой M0 содержит объекты системы во время ее исполнения.

Ниже приведен пример иерархии для конкретной системы:

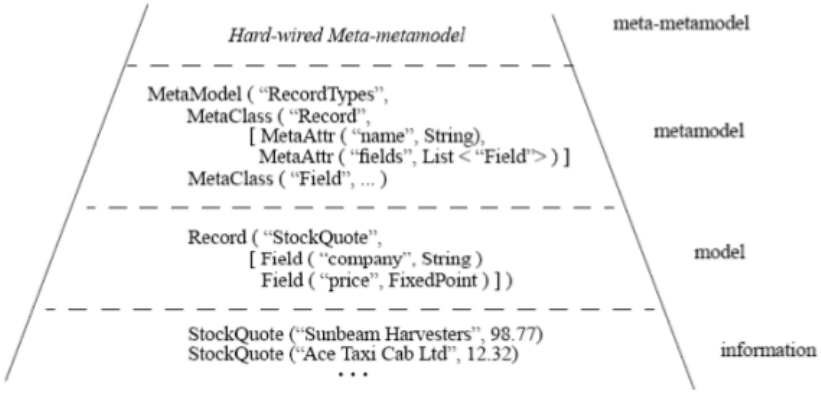

Рисунок 3.2. Пример иерархии моделей

Теоретически иерархию можно дополнить дополнительными слоями над слоем M3, но MOF позволяет рекурсивно описывать вышележащие слои, поэтому слои выше M3 не имеют смысла, так как содержат в себе ту же самую мета-метамодель.

3.2.3. Структура MOF

MOF обладает модульной структурой, каждый модуль называется *пакетом*. Пакеты можно объединять в пакеты верхнего уровня, образуя иерархию пакетов. Существует два пакета верхнего уровня - `MOF` и `UML infrastructure library`.

Стоит отметить, что MOF использует те же понятия для описания сущностей, что и UML, таким образом, между пакетами образуется следующий вид отношения:

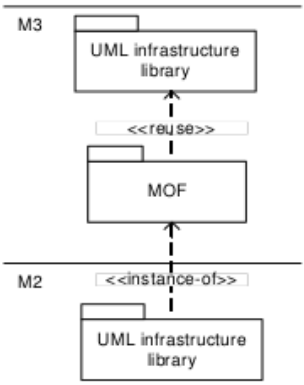

Рисунок 3.3. Взаимосвязь между пакетами MOF верхнего уровня

Данные пакеты включают в себя множество подпакетов, дерево которых приведено на рис. 3.4.

Опишем некоторые из приведенных пакетов:

Пакет Abstractions

Данный пакет содержит элементы, которые затем используются во всех других пакетах. В нем содержатся такие элементы как выражения, литералы, пространства имен, отношения и т.д.

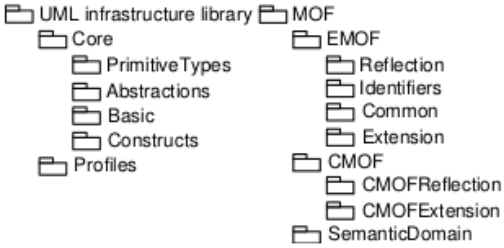

Рисунок 3.4. Дерево пакетов MOF

Пакет Basic

Пакет Basic предназначен исключительно для облегчения использования и объединяет в себе сущности из нескольких других пакетов.

Пакет Constructs

В данном пакете содержатся конкретные экземпляры классов из пакета Abstractions, например, классы различных отношений, конструкций языка и т.д.

Пакет EMOF

EMOF является объединяет в себе базовые концепции MOF, такие как Reflection (возможность доступа к свойствам описываемого объекта), Element (базовая единица метамодели, являющаяся экземпляром какого-либо класса), Common (пакет, поддерживающий коллекции экземпляров Element).

Структура данного пакета приведена на рис. 3.5:

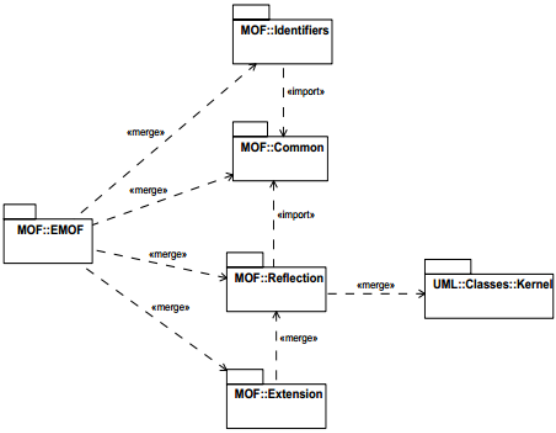

Рисунок 3.5. Структура EMOF

Пакет CMOF

Данный пакет включает в себя пакет EMOF с некоторыми дополнениями (например, он расширяет пакет Reflection, добавляя в него новые операции над сущностями).

На рис. 3.6 приведен краткий пример MOF-совместимой метамодели для описания электрических схем [14].

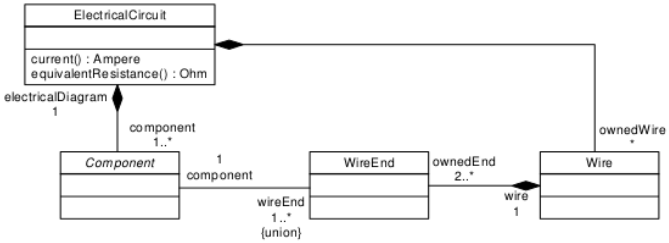

Рисунок 3.6. Метамодель для описания электрических схем

Класс `Component` является базовым классом для всех сущностей,

включая `ElectricalCircuit`, которая отображает саму электрическую схему. Класс `Wire` показывает вид отношений между объектами электрической схемы.

3.2.4. Сериализация метамодели

Для сериализации UML моделей и их передачи между различными средствами группой OMG был разработан формат XML Metadata Interchange (XMI). Но так как UML является совместимой со стандартом MOF, XMI может быть использован для сериализации любой MOF-совместимой метамодели [15].

В стандарт XMI входят следующие составляющие:

1. Набор правил DTD для отображения метамоделей в XML документ.

2. Правила описания метаданных.

3. Схему XML для XMI документов.

Пример отношения между системой на языке программирования, ее модели и сериализованной модели в XMI приведен на рис. 3.7:

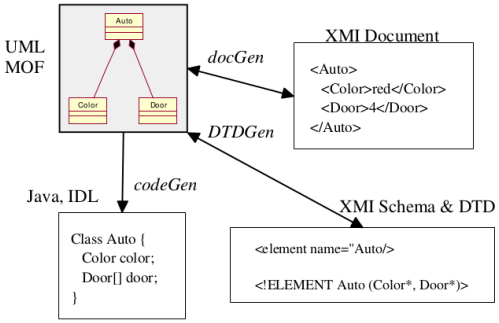

Рисунок 3.7. Отношение между MOF и XMI

Приведем пример сериализованной UML модели в формате XMI:

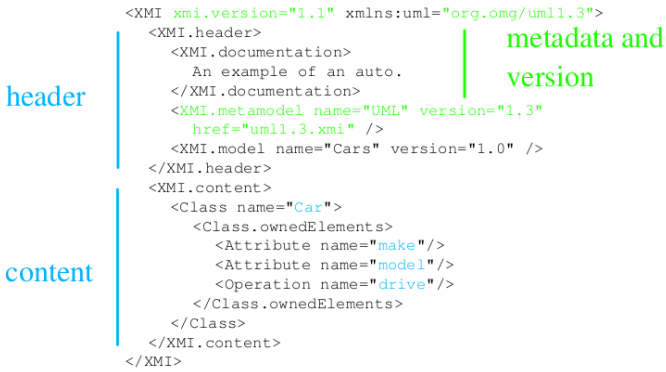

Рисунок 3.8. Пример сериализованной UML-модели

Таким образом, выбранный формат данных естественным образом ложится на разработанную архитектуру, а его стандартизированность облегчает взаимодействие между инструментальной средой и сторонними средствами. Это является неоспоримым преимуществом по сравнению с другими форматам (JSON, формат сериализации объектов JAVA и т.д.).

3.3. Проектирование архитектуры преобразователей

Задача преобразователей - отображение исходного кода программы в метамодель. Таким образом каждый преобразователь - это транслятор языка программирования, на котором написана анализируемая программа.

Как и любой транслятор, преобразователь может иметь несколько фаз выполнения [16]:

1. Лексический анализ - выделение лексем во входном тексте программы

2. Синтаксический анализ - построение синтаксического дерева по входному набору лексем в соответствии с грамматикой языка

3. Семантический анализ - проверка семантических условий языка (например, соответствие типов, наличие объявления используемой переменной и т.д.)

4. Оптимизация - преобразование промежуточного представления программы с целью повышения производительности, уменьшения размера генерируемого кода, объема используемой памяти и т.п.

5. Генерация кода - получение результата выполнения трансляции

Так как задачей преобразователя является генерация метамодели, некоторые фазы не являются необходимыми (например, фаза оптимизации). Также, можно опустить фазу семантического анализа, что сильно упростит код преобразователя, но скажется на его простоте использования, так как пользователь будет вынужден проверять корректность анализируемой системы сторонними средствами (например, при помощи компилятора языка программирования, на котором написана анализируемая система).

Получившаяся структура преобразователя изображена на рис. 3.9.

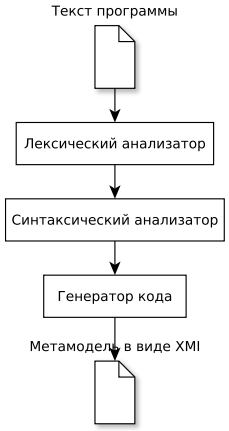

Рисунок 3.9. Архитектура преобразователя

Существует два подхода к построению лексических (лексеров) и синтаксических (парсеров) анализаторов:

1. Написание кода лексера и парсера вручную (например, парсеров, использующих метод рекурсивного спуска или метод функциональных комбинаторов).

2. Использование генераторов парсеров (например, табличных парсеров)

К достоинствам первого метода можно отнести повышенное быстродействие, пониженную ресурсоемкость и меньший объем кода лексического и синтаксического анализаторов. К недостаткам данного метода можно отнести высокую трудоемкость разработки.

Использование генераторов парсеров значительно упрощает написание, тем самым уменьшая количество возможных ошибок и дефектов. Однако сгенерированные парсеры обладают худшей производительностью и предъявляют повышенные требования к объему используемой памяти.

На основе анализа достоинств и недостатков обоих методов, было принято решение об использовании автоматической генерации кода парсеров и лексеров разрабатываемых преобразователей. В случае нехватки производительности или объема предоставляемой памяти по результатам профилирования возможна разработка новых преобразователей с использованием вручную написанных парсеров и лексеров.

Результатом работы преобразователя является сериализованная метамодель в формате XML. Так как структура XML-файла полностью отражает структуру метамодели, для ее сериализации целесообразно использовать различные фреймворки для преобразования объектов программы в формат XML.

3.4. Проектирование графического интерфейса и процедур анализа

При проектировании графической составляющей инструментальной среды необходимо решить следующие проблемы:

1. Унификация процедур анализа над метамоделью для обеспечения возможности добавления новых процедур по мере необходимости

2. Удобство отображения моделей сложных программных систем

Так как большинство процедур анализа строится на обходе структуры метамодели, то для решения первой проблемы целесообразно применить шаблон “Посетитель” (Visitor) [17]. Применение данного шаблона позволит абстрагировать метамодель от алгоритмов над ней, выделив обход ее структуры в отдельный класс.

Необходимые виды визуализации, описанные в разделе 2, имеют вид графа, поэтому для решения второй проблемы можно применить следующие способы уменьшения размера отображаемой модели системы:

1. Сворачивание отдельных узлов и примыкающих к ним дуг
2. Масштабирование всего графа относительно размера панели отображения визуализаций

3.5. Вывод

В результате была разработана архитектура инструментального средства. Вся программная система разбита на три составляющие:

1. Преобразователи, в чью задачу входит импортирование исходного кода анализируемой программы и создание сериализованной метамодели. Данный подход позволяет отделить языкозависимую часть от всей системы в целом.
2. Метамодель, являющаяся абстрактным представлением системы, над которой затем проводятся все процедуры анализа и визуализации.
3. Все процедуры анализа были вынесены в отдельную составляющую и отделены от метамодели, что позволит легко расширять комплект имеющихся процедур.

При проектировании графического интерфейса была учтена возможность извлечения крупных моделей и были приняты меры по улучшению их визуализации.

4. РАЗРАБОТКА ИНСТРУМЕНТАЛЬНОЙ СРЕДЫ

В данном разделе рассматривается реализация программной системы в соответствии с поставленными требованиями и спроектированной архитектурой.

4.1. Средства разработки

Для разработки инструментальной среды было решено использовать язык программирования Java по следующим причинам:

1. Большой выбор среди генераторов парсеров, графических библиотек, фреймворков сериализации, библиотек для работы с графами.

2. Требуется достаточно высокая производительность, так как анализируемые системы могут быть крупными, что увеличивает размер модели и сложность работы с ней.

3. Наличие опыта разработки на данном языке у автора работы, что уменьшит сроки разработки и улучшит качество программного продукта.

Разработка среды велась на языке Java версии 7.

Сборка проекта производится при помощи фреймворка для автоматизации сборки Apache Maven. Данный фреймворк позволяет декларативным образом описывать процесс сборки проекта, фокусируясь на его структуре, используя различные плагины для организации фаз построения [18]. Вся информация содержится в XML-файле под названием pom.xml (Project Object Model). Одной из отличительной особенностей Maven является наличие центрального репозитория, что сильно упрощает управление зависимостями разрабатываемого проекта. Добавляя необходимые библиотеки в pom.xml, они будут автоматически скачены из центрального репозитория и размещены в локальном для дальнейшего использования в цикле сборки проекта.

4.2. Структура проекта

В соответствии с архитектурой среды, приведенной в подразделе 3.1, проект разбит на 3 составляющих:

1. Метамодель
2. Преобразователи
3. Инструментальная среда (графический интерфейс, процедуры визуализации и анализа)

Проект разделен на три модуля, связь между которыми изображена на рис. 4.1.

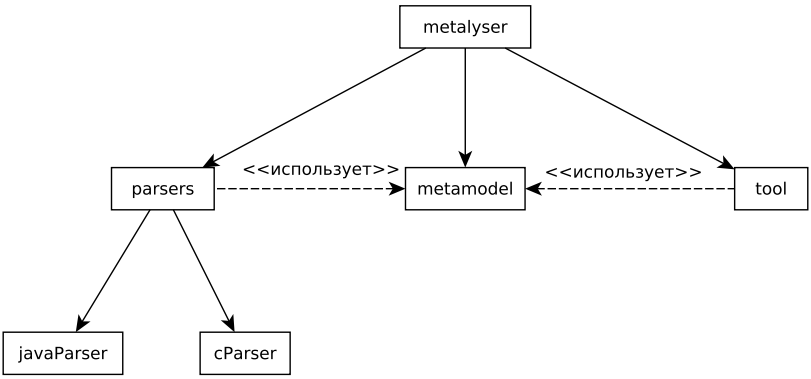

Рисунок 4.1. Структура maven-проекта

Рассмотрим подробнее приведенную структуру:

1. Модуль верхнего уровня называется `metalyser` (Meta-Analyser) и является корнем всего проекта. В нем размещаются зависимости и плагины maven, необходимые для сборки всей программной системы.
2. Модуль `parsers` является родительским модулем для преобразователей языка Java и C, поставляемых вместе с разработанной средой.

3. Модуль `javaParser` является преобразователем для языка Java. Аналогично, модуль `cParser` используется для импортирования систем, написанных на языке C.

4. Модуль `metamodel` является реализацией метамодели, описывающей анализируемую систему. Данный модуль реализован в виде библиотеки классов и не предназначен для непосредственного использования (он не содержит Main-класса для запуска), а используется в виде зависимости в остальных модулях системы.

5. Модуль `tool` является реализацией инструментальной среды.

4.3. Разработка метамодели

На основе анализа возможной архитектуры метамодели (см. подраздел 3.2) было принято решение взять за основу архитектуру, предложенную стандартом MOF [12]. Недостатком данной архитектуры является ее громоздкость: полученные метамодели охватывают широкий спектр возможных задач, что делает их достаточно объемными и сложными в использовании. Исходя из требований к системе, было решено модифицировать исходную архитектуру, а именно были убраны отдельные классы для отображения отношений между объектами. Вместо этого ссылки на зависимые объекты хранятся непосредственно в самих классах. Из-за этого теряется семантическая информация о видах отношений, но намного сокращается количество используемых классов, что сильно упрощает использование метамодели.

Таким образом, полученная метамодель не является полностью MOF-совместимой, но, если в дальнейшем развитии проекта такая совместимость будет необходима для взаимодействия со сторонними приложениями, то не составит труда модифицировать метамодель до полной совместимости.

4.3.1. Общая структура

Принимая во внимание все перечисленные замечания, была разработана соответствующая структура метамодели (краткая UML-диаграмма классов приведена на рис. 4.2). Стоит отметить, что на данной диаграмме для уменьшения ее размеров были опущены все методы, отношения между классами и некоторые наследники основных суперклассов.

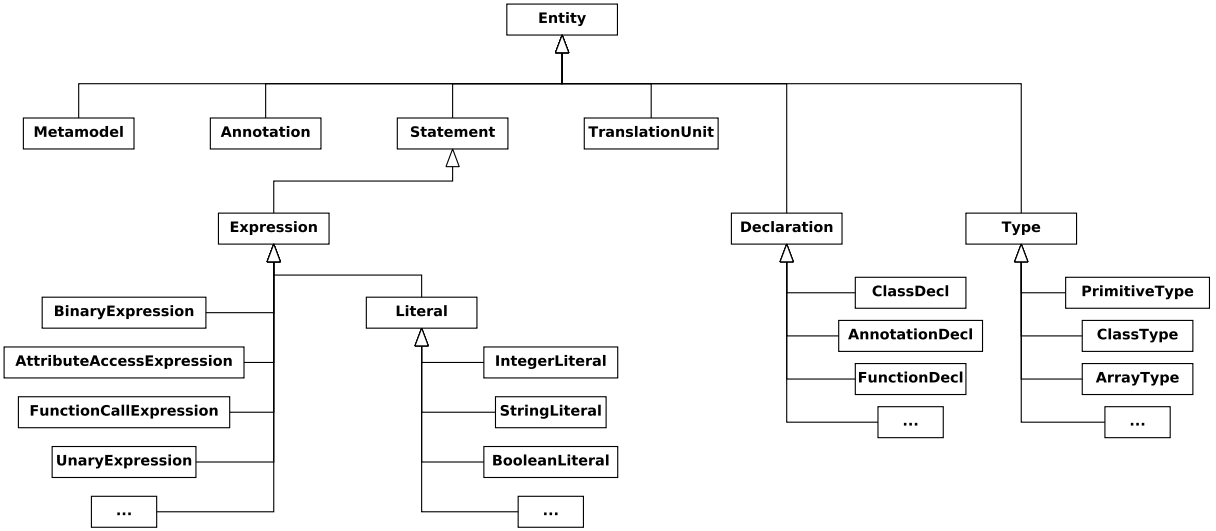

Рисунок 4.2. Упрощенная диаграмма классов метамодели

Рассмотрим подробнее некоторые из классов:

Класс Entity

Листинг 4.1. Интерфейс Entity

```
1 public interface Entity {
2     public void accept(final Visitor visitor);
3 }
```

От интерфейса `Entity` наследуются все классы метамодели. Класс содержит лишь один метод - `accept`, который необходим для обхода дерева, которое представляет из себя построенная модель. Таким образом для проведения анализа метамодели был реализован паттерн “Посетитель” (подробнее обход дерева объектов описан в подразделе 4.3.2).

Класс Metamodel

Листинг 4.2. Класс Metamodel

```
1 public class Metamodel implements Entity {
2     private final List<TranslationUnit> units;
3
4     ...
5 }
```

Данный класс является контейнером верхнего уровня для объектов типа `TranslationUnit`. Он предназначен для сборки результатов работы преобразователей над каждым из файлов с исходным кодом анализируемой программной системы.

Класс TranslationUnit

Листинг 4.3. Класс TranslationUnit

```
1 public class TranslationUnit implements Entity {
2     private final List<Import> imports;
3     private final List<Declaration> types;
4
5     ...
6 }
```

Класс `TranslationUnit` отображает одну единицу трансляции при работе преобразователя. Обычно ей соответствует один файл с исходным кодом (за исключением, например, языка C, где в каждый файл на этапе работы препроцессора вставляется текст используемых заголовочных или любых других файлов, подключенных при помощи директивы `#include`). Таким образом, обычно при трансляции всей анализируемой системы в результате работы преобразователя будет получено несколько объектов типа `TranslationUnit`, которые затем должны быть объединены в один объект типа `Metamodel`.

Класс Annotation

Листинг 4.4. Класс Annotation

```
1 public class Annotation implements Entity {
2     private final String name;
3     private final Map<String, Entity> values;
4     private final Entity value;
5
6     ...
7 }
```

Данный класс предназначен для описания аннотаций (например, в языке Java) или им подобных элементов (например, декораторов в языке Python). Если аннотация имеет параметры, то существует два варианта их хранения:

1. Если параметров несколько, то они помещаются в контейнер `values`. Необходимо учитывать, что в данном случае параметры являются именованными.

2. Если аннотация принимает один параметр, то он может быть неименованным и помещаться в поле `value`.

Класс Statement

Класс `Statement` является базовым для всех выражений в исходном коде программы. Примером наследников являются классы `IfStatement` (ветвление вида “if-else”) и `WhileStatement` (цикл “while”).

Класс Expression

Листинг 4.5. Класс Expression

```
1 public abstract class Expression implements Statement {
2     protected final Type type;
3
4     ...
5 }
```

Данный класс по своему назначению аналогичен классу `Statement`, но, в отличие от него, предназначен для описания выражений, имеющих тип. Примером таких выражений может быть обращение к переменной (класс `VariableReferenceExpression`), арифметические выражения (классы `BinaryExpression` и `UnaryExpression`), вызов функции (класс `FunctionCallExpression`) и т.д. Следует отметить, что типизированные выражения также являются обычными выражениями и могут использоваться аналогично другим наследникам класса `Statement`.

Класс Type

Данный класс предназначен для описания различного вида типов, как примитивных, так и пользовательских. Наследниками этого класса являются, например, классы `PrimitiveType` (примитивные типы) и `ArrayType` (массивы).

Класс Literal

Класс `Literal` отображает различного вида литералы в исходном коде программы. Примерами литералов могут являться целочисленные литералы (1, 42, 88), строковые литералы (“foo”, “barbaz”) и т.д.

Класс Declaration

Листинг 4.6. Класс Declaration

```
1 public abstract class Declaration implements Entity {
2     protected final String name;
3     protected Visibility visibility;
4     protected final List<String> modifiers;
5     protected final List<Annotation> annotations;
6
7     ...
8 }
```

Класс `Declaration` является суперклассом для объявлений типов и глобальных переменных. Каждое объявление может быть помечено аннотациями (`annotations`) или различными модификаторами (например, `final` в языке Java или `const` в C). Один вид модификаторов, а именно модификаторы доступа, вынесены в тип `Visibility`. Это сделано для облегчения подсчета метрик, а также потому что в большинстве языков программирования назначение этих модификаторов совпадает (поэтому данное допущение не ухудшит универсальность метамодели). Примерами наследников данного класса являются классы `ClassDecl` (объявление класса) и `FunctionDecl` (объявление функции).

4.3.2. Проведение операций над метамоделью

Существует два подхода к реализации паттерна “Посетитель” в языке Java:

1. “Классический” подход
2. Подход с использованием рефлексии в языке Java (Java Reflection API [19])

Рассмотрим оба варианта подробнее:

“Классический” подход

В данном случае в классе-“посетителе” необходимо реализовывать методы `visit` для каждого типа иерархии объектов. К недостаткам данного подхода можно отнести очень малую гибкость - при изменении иерархии классов, над которой совершается обход, аналогичные изменения необходимо вносить в каждый из классов-“посетителей”.

Подход с использованием рефлексии

Подход с использованием рефлексии использует метод языка Java `Java.lang.Class.getMethod()` для поиска метода `visit` для типа объекта-параметра. Достоинствами данного метода является отсутствие необходимости реализации метода `visit` для каждого возможного класса из иерархии: пользователь должен реализовать метод только для тех классов, которые необходимы для той или иной процедуры, производимой над этой иерархией. Недостатком данного подхода является пониженное быстродействие по сравнению с "классическим" подходом (из-за низкой производительности Reflection API).

Так как метамодель, в соответствии с требованиями, должна быть расширяемой и позволять легко добавлять новые классы к уже существующим, то было принято решение использовать подход с использованием рефлексии.

Исходя из вышесказанного был разработан интерфейс `Visitor`:

Листинг 4.7. Базовый класс для обхода метамодели

```
1 public interface Visitor {
2     public void dispatch(final Entity entity);
3     public void navigate(final Entity entity);
4 }
```

Метод `dispatch` вызывается на корневом объекте метамодели (чаще всего это экземпляр класса `Metamodel`, но это условие не является обязательным) и инициирует процедуру обхода. Метод `navigate` предназначен для определения порядка обхода вложенных элементов метамодели: для каждого класса, унаследованного от `Entity`, необходимо переопределить метод `accept`, в котором пользователь может задавать, в каком порядке будут посещены поля этого класса. Также поддерживается навигация непосредственно из класса, реализующего интерфейс `Visitor`: для этого необходимо переопределить метод `navigate`, чтобы он оставался пустым.

Примером реализации интерфейса `Visitor` является класс `VisitorAdapter`:

Листинг 4.8. Реализация интерфейса Visitor

```
1 public class VisitorAdapter implements Visitor {
2     @Override
3     public void dispatch(final Entity entity) {
4         if (entity == null) return;
5         try {
6             final Method m = getClass().getMethod("visit",
```

```
7                   new Class<?>[] { entity.getClass() });
8               m.invoke(this, new Object[] { entity });
9           } catch (NoSuchMethodException ex) {
10              // протоколирование ошибки
11          } catch (IllegalAccessException | IllegalArgumentException |
                InvocationTargetException ex) {
12              // протоколирование ошибки
13          }
14          navigate(entity);
15      }
16
17      @Override
18      public void navigate(Entity entity) {
19          entity.accept(this);
20      }
21  }
```

Переопределенный метод `dispatch` работает следующим образом:

1. Получение метода `visit` для класса объекта, переданного в качестве параметра.

2. В случае отсутствия такого метода, происходит запись сообщения с соответствующей информацией в лог, но ход программы не прерывается.

3. Вызов метода на объекте-параметре.

4. Вызов метода `navigate` для дальнейшего обхода объектов метамодели. В данном случае используется навигация по-умолчанию, описанная в классах метамодели.

4.3.3. Выбор фреймворка для сериализации

Для языка Java существует множество фреймворков для генерации XML-файлов, самыми популярными из них являются:

- Java Architecture for XML Binding (JAXB)
- Simple
- XStream

Во всех фреймворках описание правил сериализации производится при помощи аннотирования необходимых элементов классов.

Фреймворк JAXB

JAXB представляет собой стандарт, описывающий преобразование Java объектов в XML (маршалинг) и обратно (демаршалинг) [20]. Так как JAXB является стандартом, то существует несколько его реализаций, например, Metro, EclipseLink MOXy, JaxMe. Отличительной особенностью данного стандарта является наличие его реализации в составе Java SE, начиная с версии 6, тем самым исключается необходимость внедрения дополнительных зависимостей в проект.

Пример использования аннотаций JAXB:

Листинг 4.9. Пример использования фреймворка JAXB

```
@XmlRootElement
public class Customer {
    String name;
    int age;
    int id;

    @XmlElement
    public void setName(String name) {
        this.name = name;
    }

    @XmlElement
    public void setAge(int age) {
        this.age = age;
    }

    @XmlAttribute
    public void setId(int id) {
        this.id = id;
    }

    ...
}
```

XML-файл после сериализации:

Листинг 4.10. Полученный XML-файл

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<customer id="100">
    <age>23</age>
    <name>Alexander</name>
</customer>
```

Фреймворк Simple

Фреймворк Simple [21] организован таким образом, чтобы как можно больше снизить необходимость конфигурирования, что умень-

шает количество используемых аннотаций в коде. Еще одной отличительной особенностью данного фреймворка является наличие возможности десереализации неизменяемых объектов, что актуально для большинства классов разработанной метамодели.

Пример использования:

Листинг 4.11. Пример использования фреймворка Simple

```
@Root
public class Example {
   @Element
   private String text;

   @Attribute
   private int index;

   ...
}
```

Полученный XML-файл:

Листинг 4.12. Полученный XML-файл

```
<example index="123">
   <text>Example message</text>
</example>
```

Фреймворк XStream

XStream по своим возможностям практически ничем не отличается от уже описанной раннее реализации JAXB. К недостаткам данного фреймворка можно отнести отсутствие подробной документации.

Пример использования:

Листинг 4.13. Пример использования фреймворка XStream

```
@XStreamAlias("message")
class RendezvousMessage {

    @XStreamAlias("type")
    @XStreamAsAttribute
    private int messageType;

    @XStreamImplicit(itemFieldName="part")
    private List<String> content;

    ...
}
```

Полученный XML-файл:

Листинг 4.14. Полученный XML-файл

```
<message type="15">
    <part>firstPart</part>
    <part>secondPart</part>
</message>
```

Вывод

Таким образом, после проведения анализа, для сериализации метамодели был выбран фреймворк Simple. Ключевым фактором является возможность данного фреймворка сериализовывать неизменяемые объекты, а наличие сторонней зависимости не является проблемой, благодаря системе сборки Maven.

Пример использования фреймворка Simple в разработанной метамодели:

Листинг 4.15. Пример использования фреймворка Simple в реализации метамодели

```
public abstract class Declaration implements Entity {
    @Element(required = false)
    protected final String name;
    @Element
    protected Visibility visibility;
    @ElementList(required = false)
    protected final List<String> modifiers;
    @ElementList(required = false)
    protected final List<Annotation> annotations;

    public Declaration(
        @Element(name = "name")
        final String name,
        @Element(name = "visibility")
        final Visibility visibility,
        @ElementList(name = "modifiers")
        final List<String> modifiers,
        @ElementList(name = "annotations")
        final List<Annotation> annotations) {

        ...
    }

    ...
}
```

Аннотация `@Element` обозначает один тэг в итоговом XML-документе, где аргумент `required` указывает на то, что данный тэг не является обязательным. Аннотация `@ElementList` описывает коллекцию тэгов. Так как класс `Declaration` является неизменяемым, то

для десереализации необходимо предоставить конструктор, позволяющий инициализировать все поля класса, при этом каждый параметр необходимо пометить соответствующей аннотацией с указанием имени поля.

В данном проекте используется версия Simple 2.7.1, выпущенная в 2013 году. Пример сериализованной метамодели приведен в приложении А.

4.4. Разработка преобразователей

При разработке архитектуры преобразователей (подраздел 3.3) было принято решение об использовании генераторов парсеров для проведения лексического и синтаксического анализа при построении модели анализируемой программы.

4.4.1. Выбор генератора лексического и синтаксического анализаторов

Наиболее популярными генераторами парсеров (которые активно разрабатываются и поддерживатся) на языке Java на данный момент являются следующие программные средства:

- ANTLR (ANother Tool for Language Recognition)
- JavaCC (Java Compiler Compiler)
- Jparsec

Фреймворк ANTLR

ANother Tool for Language Recognition - генератор синтаксических и лексических анализаторов. Данное средство генерирует нисходящий анализатор на основе LL(*) грамматики. Достоинствами данного генератора является комбинированная грамматика (единая для парсера и лексера), наличие среды разработки (ANTLRWorks), подробная диагностика ошибок, а так же способность разбирать леворекурсивные грамматики (которые обычно недопустимы при построении LL-анализаторов) [22].

Для описания грамматики используется форма, близкая к РБНФ:

Листинг 4.16. Пример грамматики, используемой в средстве ANTLR

```
1  grammar T; //имя грамматики
2  //нетерминальные символы:
3  msg : 'name' ID ';'
4      {
5          System.out.println("Hello, " + $ID.text + "!");
6      } ;
7  //терминальные символы
8  ID: 'a'..'z' + ;//произвольное (но >=1) количество букв
9  WS: (' ' |'\n' |'\r' )+; // пробел, перенос строки, табуляция
```

Фреймворк JavaCC

Как и средство ANTLR, JavaCC генерирует парсер на основе рекурсивного спуска, но с использованием LL(1) грамматики [23], что увеличивает производительность сгенерированного кода, но также увеличивает сложность написания грамматики.

Пример грамматики:

Листинг 4.17. Пример грамматики, используемой в средстве JavaCC

```
1  SKIP:  { " " | "\t" | "\n" | "\r"                      }
2  TOKEN: { "(" | ")" | "+" | "*" | <NUM: (["0"-"9"])+> }
3
4  void S(): {} { E() <EOF>            }
5  void E(): {} { T() ("+" T())*       }
6  void T(): {} { F() ("*" F())*       }
7  void F(): {} { <NUM> | "(" E() ")" }
```

Фреймворк Jparsec

Jparsec так же используется для генерации парсера на основе рекурсивного спуска. Главной отличительной особенностью данного генератора является тот факт, что он использует комбинаторы синтаксического анализа. Это значит, что он не использует отдельный файл с грамматикой в форме РБНФ, на основе которого генерируется код парсера на языке Java. Вместо этого каждому нетерминальному символу грамматики ставится в соответствие объект языка Java, а операции по составлению данного символа - альтернатива (символ “|” в РБНФ), зависимость (последовательность лексем) и т.д., описываются при помощи методов этого объекта.

Приведем пример парсера, использующего данный фреймворк:

Листинг 4.18. Пример парсера, использующего фреймворк Jparsec

```
private static final Terminals OPERATORS =
    Terminals.operators("+", "-", "*", "/", "(", ")");

static final Parser<?> TOKENIZER =
    Parsers.or(Terminals.DecimalLiteral.TOKENIZER,
               OPERATORS.tokenizer());

static Parser<?> term(String... names) {
    return OPERATORS.token(names);
}

static final Parser<BinaryOperator> WHITESPACE_MUL =
    term("+", "-", "*", "/").not().retn(BinaryOperator.MUL);

static <T> Parser<T> op(String name, T value) {
    return term(name).retn(value);
}

static Parser<Double> calculator(Parser<Double> atom) {
    Parser.Reference<Double> ref = Parser.newReference();
    Parser<Double> unit =
        ref.lazy().between(term("("), term(")")).or(atom);
    Parser<Double> parser = new OperatorTable<Double>()
        .infixl(op("+", BinaryOperator.PLUS), 10)
        .infixl(op("-", BinaryOperator.MINUS), 10)
        .infixl(op("*", BinaryOperator.MUL).or(WHITESPACE_MUL), 20)
        .infixl(op("/", BinaryOperator.DIV), 20)
        .prefix(op("-", UnaryOperator.NEG), 30)
        .build(unit);
    ref.set(parser);
    return parser;
}
```

Использование данного подхода исключает необходимость фазы генерации кода при сборке проекта, а разработанный код в несколько раз меньше по объему, чем сгенерированный. К недостаткам данного фреймворка можно отнести сложную реализацию необходимой грамматики и неудовлетворительное информирование об ошибках при проведении разбора.

Вывод

Таким образом, самым совершенным средством для разработки синтаксических и лексических анализаторов является фреймворк ANTLR, в силу простоты использования и неплохой эффективности полученных анализаторов.

В данном проекте используется версия ANTLR 4.2.2, выпущенная в 2014 году.

4.4.2. Правила использования генератора парсеров ANTLR

Общая структура файла для описания грамматики ANTLR выглядит следующим образом:

Листинг 4.19. Структура грамматики ANTLR

```
grammar Name; // имя грамматики
options {...}
import ... ;
tokens {...}
@actionName {...}

rule1 // правила грамматики
...
ruleN
```

Секция `import` предназначена для импортирования других файлов с правилами грамматики, тем самым позволяя создавать иерархичную структуру ANTLR-проекта.

Секция `tokens` позволяет явно задавать типы лексем лексического анализатора. На практике это означает, что будет сгенерировано перечисление (enum), которое затем может быть использовано в коде парсера для отсылке к той или иной лексеме.

Правила вида "`@действие {...}`" позволяют использовать набор определенных действий, поддерживаемых ANTLR. Примерами являются действие `@header` для описания заголовочной части сгенерированного кода парсера и действие `@after` для генерации кода, который будет выполнен непосредственно перед вызовом определенного правила грамматики. Пример использования приведен в листинге 4.20:

Листинг 4.20. Использование действий ANTLR

```
grammar Count;
// объявление пакета, в котором располагается код парсера
@header {
package foo;
}

@members {
int count = 0; // объявление члена класса парсера
}

list
// вывод количества чисел во входной последовательности лексем
@after {System.out.println(count + " ints");}
 : INT {count++;} (',' INT {count++;} )*
 ;

INT : [0-9]+ ;
```

Секция `options` задает параметры генерации кода, например, язык на котором будет сгенерирован полученный парсер.

ANTLR позволяет создавать смешанные грамматики, т.е. в одном и том же файле задаются как правила синтаксического анализа, так и лексического. Правила разбора лексем имеют вид "`ИМЯ : правило`", где правая часть задает название правила, а левая - правило его разбора. Аналогичный формат имеют правила для нетерминальных символов с той лишь разницей, что имена лексем задаются заглавными буквами, а нетерминалов - прописными.

4.4.3. Реализация преобразователя для языков Java

В основу реализации парсера легла грамматика языка Java, поставляемая вместе с исходным кодом фреймворка ANTLR. Грамматика была модифицирована для упрощения построения модели анализируемой программы (текст грамматики приведен в приложении Б).

Код построения модели программы интегрирован в текст грамматики. ANTLR делается следующим образом (в листинге 4.21 приведен пример построения объекта, отображающего объявление класса в исходном коде анализируемой программы):

Листинг 4.21. Пример построения модели

```
1  classDeclaration
2  returns [ClassDecl result]
3  locals [
4      List<TemplateDecl> templates = Collections.emptyList(),
5      List<Type> inherits = new LinkedList<>()
6  ]
7  :   'class' Identifier
8      (   typeParameters
9          { $templates = $typeParameters.result; }
10     )?
11     (   'extends' type
12         { $inherits.add($type.result); }
13     )?
14     (   'implements' typeList
15         { $inherits.addAll($typeList.result); }
16     )?
17     classBody
18     {
19         $result = new ClassDecl($Identifier.text,
20                                 $templates,
21                                 $inherits,
22                                 $classBody.result);
23     };
```

После генерации кода лексического и синтаксического анализаторов (для ANTLR существует плагин для системы сборки Maven, поэтому генерация кода интегрирована в процесс сборки проекта), обращение к сгенерированным классам выглядит следующим образом:

Листинг 4.22. Использование сгенерированных классов

```
1       ANTLRInputStream in =
2           new ANTLRInputStream(new FileInputStream(input));
3       JavaGrammarLexer lexer = new JavaGrammarLexer(in);
4       CommonTokenStream tokens = new CommonTokenStream(lexer);
5       JavaGrammarParser parser = new JavaGrammarParser(tokens);
6       result.add(parser.compilationUnit().result);
```

Разработанный преобразователь поддерживает стандарт языка Java версии 7.

4.4.4. Реализация преобразователя для языка C

Аналогичным образом с использованием фреймворка ANTLR был разработан парсер для языка C. Данный преобразователь является прототипом и предназначен исключительно для демонстрации возможности метамодели описывать системы, написанные на различных языках программирования. Данный прототип обладает существенными ограничениями, в частности, он не позволяет обрабатывать макросы языка C, а, следовательно, и системы, состоящие из нескольких файлов (поскольку в C используется текстуальное включение зависимых файлов посредством препроцессора). Также отсутствует поддержка GNU расширений языка C.

Для реализации парсера так же использовалась модифицированная грамматика языка C, поставляемая вместе с фреймворком ANTLR. Текст грамматики приведен в приложении B.

4.5. Разработка графического интерфейса и процедур анализа

Разработка графического интерфейса велась с использованием библиотеки Swing. Интерфейс поддерживает следующие функции:

1. Загрузка файла с метамоделью
2. Визуализация моделей (AST и CFG)

3. Визуализация UML-диаграммы классов

4. Отображение результата по подсчету метрик

Для обеспечения перечисленных функций интерфейса были разработаны соответствующие процедуры анализа, которые будут рассмотрены в данном разделе.

4.5.1. Выбор библиотеки для визуализации графов

Так как большинство моделей программ имеют структуру в виде графа, то центральной составляющей для интерфейса инструментальной среды является библиотека визуализации графов.

Рассмотрим следующие библиотеки визуализации графов:

1. yFiles

2. JGraphX

3. GraphStream

Библиотека yFiles

Данная библиотека предоставляет API для построения, визуализации и анализа графов. Она разделяется на три части:

- Basic - пакет, состоящий из набора основных классов для построения графов и структур данных для удобства программирования.

- Layout - обширная группа методов для размещения вершин графа на экране.

- Viewer - данная группа классов предназначена для облегчения редактирования отображенного графа.

Данная библиотека является очень мощным средством и позволяет решать широкий круг задач. Единственным недостатком является закрытая лицензия и дорогостоящая подписка.

Библиотека JGraphX

Библиотека JGraphX так же предназначена для визуализации графов, но обладает более скромными возможностями, чем yFiles (например, малый набор методов размещения вершин). Однако, в отличие от yFiles, данная библиотека обладает свободной лицензией и является бесплатной. Достоинствами данной библиотеки также является наличие гибкой системы настройки внешнего вида вершин, что актуально для решения задачи визуализации моделей и построения диаграмм. К недостаткам данной библиотеки можно отнести отсутствие подробной документации.

Библиотека GraphStream

Данная библиотека в основном предназначена для анализа структуры графа и обладает широким рядом алгоритмов анализа (алгоритмы поиска, извлечения минимального остовного дерева и т.д.). Недостатками GraphStream является практически полное отсутствие возможности настройки внешнего вида вершин и дуг, что делает ее непригодной для использования в проекте.

Вывод

В результате анализа было принято решение об использовании библиотеки JGraphX. Она является свободным ПО и обладает полным набором необходимых функций для решения задач визуализации моделей, а именно:

- Функции автоматического размещения вершин
- Возможность настройки внешнего вида вершин и дуг
- Функции масштабирования крупных графов

4.5.2. Разработка процедуры построения CFG

Для построения CFG используется класс `CfgDrawVisitor`:

Листинг 4.23. Класс CfgDrawVisitor

```
1  public class CfgDrawVisitor extends DrawVisitor {
2      ...
3  }
```

Класс `DrawVisitor` является оберткой над классом `VisitorAdapter` (см. подразд. 4.3.2) и содержит в себе текущий граф:

Листинг 4.24. Класс DrawVisitor

```
1  public class DrawVisitor extends VisitorAdapter {
2      protected final Graph graph;
3
4      ...
5  }
```

Алгоритм работы процедуры построения выглядит следующим образом:

1. Вызов метода `dispatch` на объекте класса `Metamodel` - начало обхода метамодели.
2. Если объект-параметр является сущностью, отображающей управляющую конструкцию, то вызывается соответствующий метод отрисовки. Управляющими конструкциями являются:
   - Ветвления - `IfStatement`, `SwitchStatement` и т.д.
   - Циклы - `WhileStatement`, `ForStatement` и т.д.
   - Операции безусловного перехода - `BreakStatement`(Java), `GotoStatement`(C) и т.д.
   - Соответствующим образом отрисовываются блоки вида try-catch-finally.
3. Иначе - создание вершины графа и ее соединение с предыдущей вершиной. Внутри вершины отображается текстовое представление объекта-параметра.
4. После завершения обхода происходит расстановка вершин графа при помощи класса `mxHierarchicalLayout` из библиотеки JGraphX.

4.5.3. Разработка процедуры построения AST

За отрисовку AST отвечает класс `AstDrawVisitor`:

```
1  public class AstDrawVisitor extends DrawVisitor {
2      ...
3  }
```

Стоит отметить, что метамодель не позволяет получить синтаксическое дерево исходной системы, так как не содержит достаточно информации о синтаксисе языка, на котором она была написана. Однако большинство конструкций во многих языках очень похожи, поэтому актуальность данной процедуры сохраняется даже в этом случае.

Визуализация AST выполняется по следующему алгоритму:

1. Вызов метода `dispatch` на объекте класса `Metamodel` - начало обхода метамодели.

2. Для каждого объекта-параметра вызывается соответствующая процедура отрисовки. При этом, в зависимости от семантики атрибутов объектов, они рисуются либо как терминальные, либо как нетерминальные вершины. Чаще всего нетерминальными вершинами являются атрибуты, тип которых принадлежит к классам метамодели.

3. Все дочерние терминальные и нетерминальные вершины графа соединяются с родительскими.

Для организации графа в виде дерева используется алгоритм библиотеки JGraphX, за который отвечает класс `mxCompactTreeLayout`.

4.5.4. Разработка процедуры построения UML-диаграммы классов

За визуализацию UML-диаграммы классов отвечает класс `ClassDiagramDrawVisitor`:

```
1  public class ClassDiagramDrawVisitor extends DrawVisitor {
2      ...
3  }
```

Так как задача построения диаграммы классов не входила в число основных и при разработке метамодели было принято решение об удалении некоторой семантической информации в виде отдельных классов отношений, то все виды взаимоотношений между классами анализируемой системы получить невозможно. Поддерживаются следующие виды отношений:

1. Обобщение (наследование)

2. Ассоциации (мощностью 1)

3. Ассоциации мощности два и более было решено рисовать в виде композиции. К сожалению, выразительности метамодели, да и самих языков программирования, недостаточно, чтобы по исходному коду различить ассоциацию, композицию и агрегацию.

4.5.5. Разработка процедуры подсчета метрик

Для демонстрации расчета метрик было решено использовать метрики Абреу [24], так как они довольно просты для расчета и предназначены для оценки объектно-ориентированных систем, что удобно продемонстрировать на примере Java-программ.

К данному классу метрик следующие метрики:

1. Фактор закрытости метода (MHF) - показывает долю скрытых методов в программе. Вычисляется по формуле:

$$MHF = \frac{\sum_{1..N}(Mh_i)}{\sum_{1..N}(Mh_i + Mv_i)} \tag{4.1}$$

   - Mh_i - число скрытых неунаследованных методов класса i
   - Mv_i - число видимых неунаследованных методов класса i

2. Фактор закрытости свойства (AHF) - показывает долю скрытых свойств в программе. Вычисляется по формуле:

$$AHF = \frac{\sum_{1..N}(Ah_i)}{\sum_{1..N}(Ah_i + Av_i)} \tag{4.2}$$

   - Ah_i - число скрытых неунаследованных методов класса i
   - Av_i - число видимых неунаследованных методов класса i

3. Фактор наследования метода (MIF) - показывает долю унаследованных непереопределенных методов в программе:

$$MIF = \frac{\sum_{1..N}(MI_i)}{\sum_{1..N}(MN_i + MI_i + MO_i)} \tag{4.3}$$

   - MI_i - число унаследованных непереопределенных методов класса i

- MN_i - число новых методов класса i
- MO_i - число унаследованных переопределенных методов класса i

4. Фактор наследования свойства (AIF) - показывает долю унаследованных непереопределенных свойств в программе. Вычисляется по формуле:

$$AIF = \frac{\sum_{1..N}(AI_i)}{\sum_{1..N}(AN_i + AI_i + AO_i)} \tag{4.4}$$

   - AI_i - число унаследованных непереопределенных свойств класса i
   - AN_i - число новых свойств класса i
   - AO_i - число унаследованных переопределенных свойств класса i

5. Фактор полиморфизма (POF)

$$POF = \frac{\sum_{1..N}(MO_i)}{\sum_{1..N}(MN_i + D_i)} \tag{4.5}$$

   - MN_i - число новых методов класса i
   - MO_i - число унаследованных переопределенных методов класса i
   - D_i - количество потомков класса i

6. Фактор сцепления (COF) - определяет долю пар классов, связанных отношением “клиент-поставщик”:

$$COF = \frac{\sum_{i \in 1..N} \sum_{j \in 1..N}(C_{ij})}{N * (N - 1)} \tag{4.6}$$

   - $C_{ij} = 1$, если класс i имеет ссылку на класс j

Метрики высчитываются вместе с построением UML-диаграммы классов, что позволяет интерактивно отображать их на экране при щелчке на каком-либо классе.

4.5.6. Разработка графического интерфейса

Главный класс интерфейса наследуется от класса `JFrame` и разделяется на три области, изображенных на рис. 4.3:

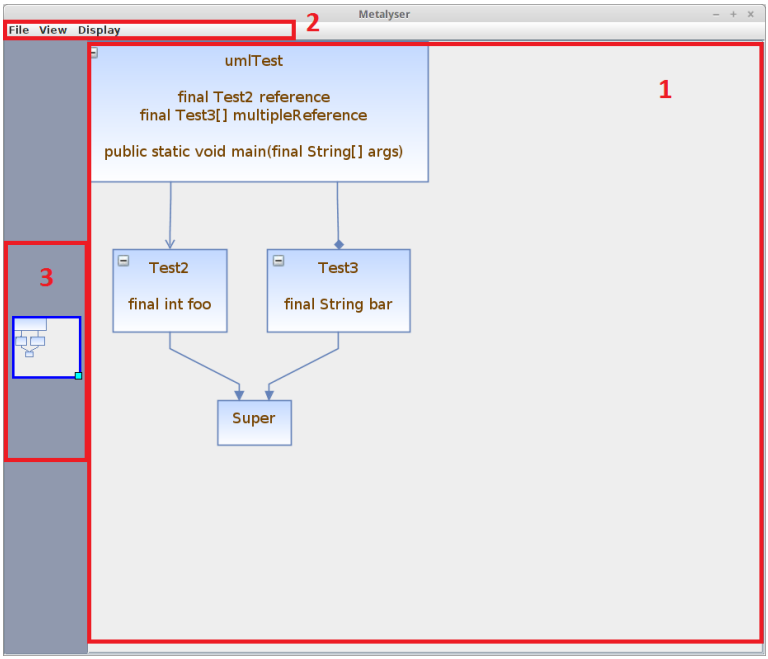

Рисунок 4.3. Структура графического интерфейса

1. Область визуализации моделей - здесь отображаются все визуализации.

2. Меню - меню позволяет открывать файл с моделью и выбирать вид визуализации.

3. Миникарта - облегчает навигацию по большим графам.

4.6. Вывод

В результате была разработана инструментальная среда в соответствии со сформулированными требованиями.

Была разработана метамодель на основе стандарта MOF, но с небольшими изменениями. Для сериализации метамодели в формат XML используется фреймворк Simple.

При помощи фреймворка для построения лексических и синтаксических анализаторов ANTLR были разработаны преобразователи для языков Java и C.

Графический интерфейс пользователя разрабатывался с применением библиотек Swing и JGraphX. Интерфейс позволяет строить визуализации AST, CFG, UML-диаграмм классов и производить подсчет метрик Абреу.

5. ТЕСТИРОВАНИЕ ИНСТРУМЕНТАЛЬНОЙ СРЕДЫ

В данном разделе ставится задача проведения тестирования разработанной программной системы на предмет наличия дефектов и соответствия функциональным и нефункциональным требованиям.

5.1. Функциональное тестирование

Функциональное тестирование - это тестирование ПО в целях проверки реализуемости функциональных требований. Для решения данной задачи необходимо протестировать каждый элемент системы по отдельности и взаимодействие этих элементов в целом. Так как метамодель лишь хранит данные и не содержит методы их обработки, ее тестирование проводилось совместно с графическим интерфейсом и процедурами анализа.

5.1.1. Тестирование преобразователей

Так как большая часть преобразователей - это код, сгенерированный при помощи фреймворка ANTLR, модульное тестирование его функций не проводилось. Для тестирования работоспособности использовался скрипт, запускающий соответствующий преобразователь для разбора какой-либо системы. В качестве набора систем для тестирования использовалось несколько десятков проектов с сайта github.com. Правильность сериализованной модели проверялось при помощи графических средств разработанной инструментальной среды (см. подраздел 5.1.2).

5.1.2. Тестирование визуализации AST и CFG

В силу невозможности автоматизации проверки правильности отображения моделей и метрик тестирование графического интерфейса проводилось вручную.

Алгоритм работы с системой выглядит следующим образом:

1. Передача исходного кода анализируемой системы преобразователю для соответствующего языка программирования.

2. Получение экземпляра метамодели и ее сериализация в формат XML.

3. Десереализация и загрузка метамодели в инструментальную среду.

4. Вызов процедур визуализации и анализа в ответ на действия пользователя.

Для тестирования графического интерфейса был разработан набор небольших программ для отображения основных конструкций языка. Ниже приведено несколько этапов протокола испытаний:

Тест №1

Данный тест предназначен для проверки визуализации оператора условного ветвления. Исходный код программы:

Листинг 5.1. Тестовая программа

```
1  class Condition {
2      public static void main(String[] args) {
3          boolean learning = true;
4
5          if (learning) {
6              System.out.println("Java programmer");
7          } else {
8              System.out.println("What are you doing here?");
9          }
10     }
11 }
```

Визуализация графа потока управления. Как видно из рис. 5.1 операция ветвления была отображена верно.

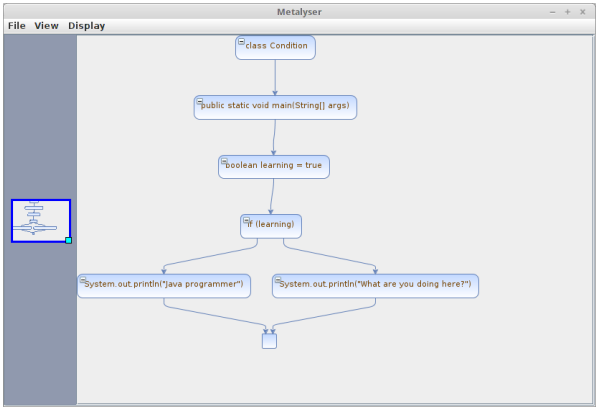

Рисунок 5.1. Визуализация CFG тестового примера

Визуализация абстрактного синтаксического дерева:

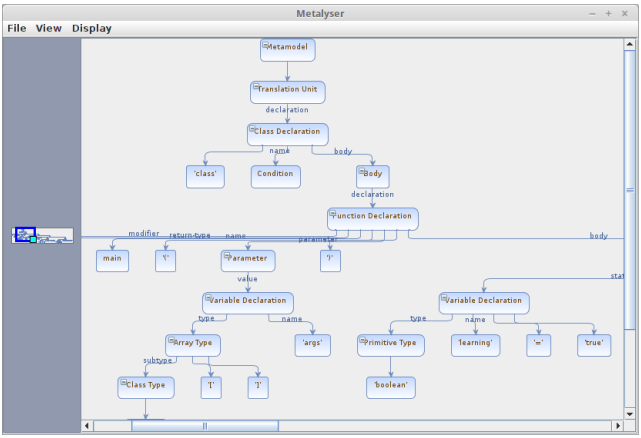

Рисунок 5.2. Визуализация AST тестового примера

Тест №2

Данный тест позволяет проверить правильность построения циклов. Исходный код программы:

Листинг 5.2. Тестовая программа

```
1 class Integers {
2     public static void main(String[] arguments) {
3         int c;
4         for (c = 1; c <= 10; c++) {
5           System.out.println(c);
6         }
7     }
8 }
```

Визуализация CFG:

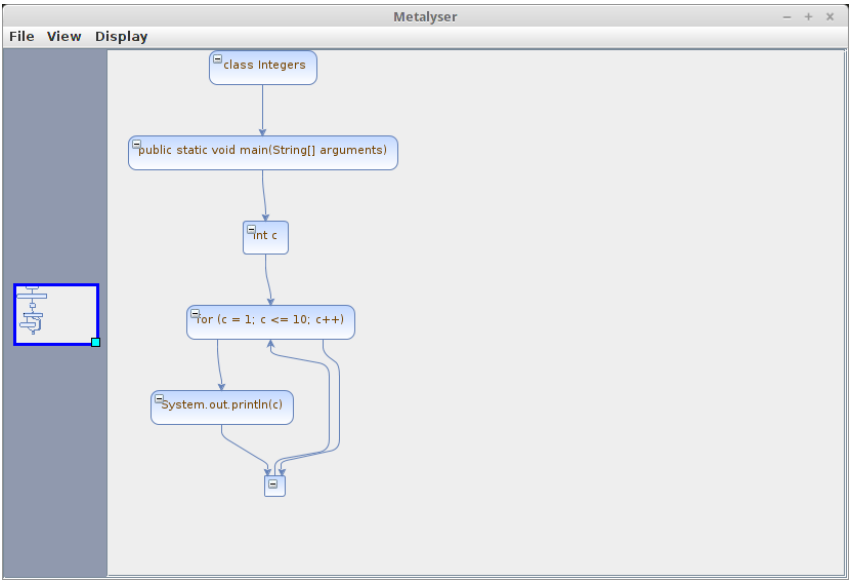

Рисунок 5.3. Визуализация CFG тестового примера

Визуализация абстрактного синтаксического дерева:

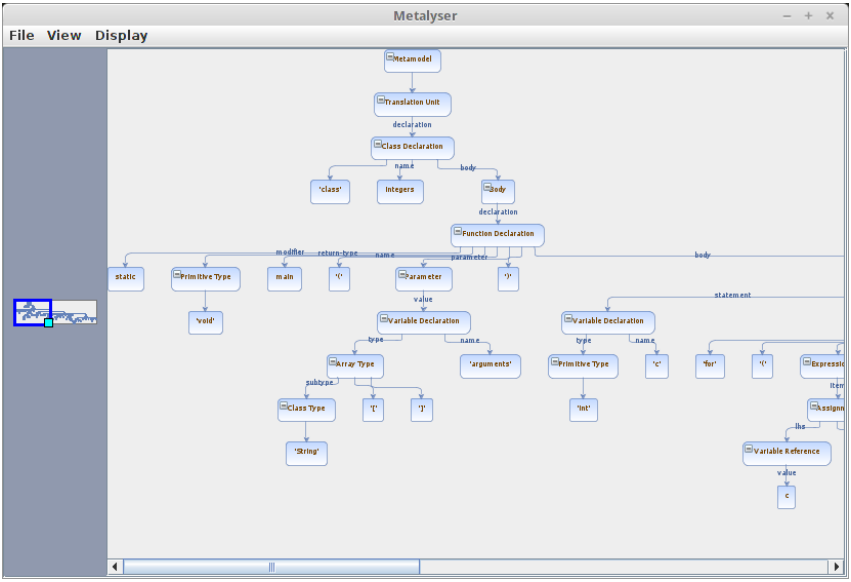

Рисунок 5.4. Визуализация AST тестового примера

5.1.3. Тестирование построения UML-диаграмм и метрик

Для проверки правильно визуализации UML-диаграммы классов был разработан следующий тестовый случай:

Листинг 5.3. Тестовая программа

```
1  public class umlTest extends umlSuper {
2      private final Test2 reference;
3      private final Test3[] multipleReference;
4
5      private int private1() {}
6      private int private2() {}
7      private int private3() {}
8
9      public void public1() {}
10
11     public static void main(final String[] args) {
12         System.out.println("Hello, world!");
13     }
14 }
15
16 class umlSuper {
```

```
17      protected int x;
18      public void public1() {}
19      public void public2() {}
20      public void public3() {}
21  }
22
23  class Test2 extends Super {
24      private final int foo;
25  }
26
27  class Test3 extends Super {
28      private final String bar;
29  }
30
31  class Super {
32  }
```

Соответствующая ему UML-диаграмма классов выглядит следующим образом:

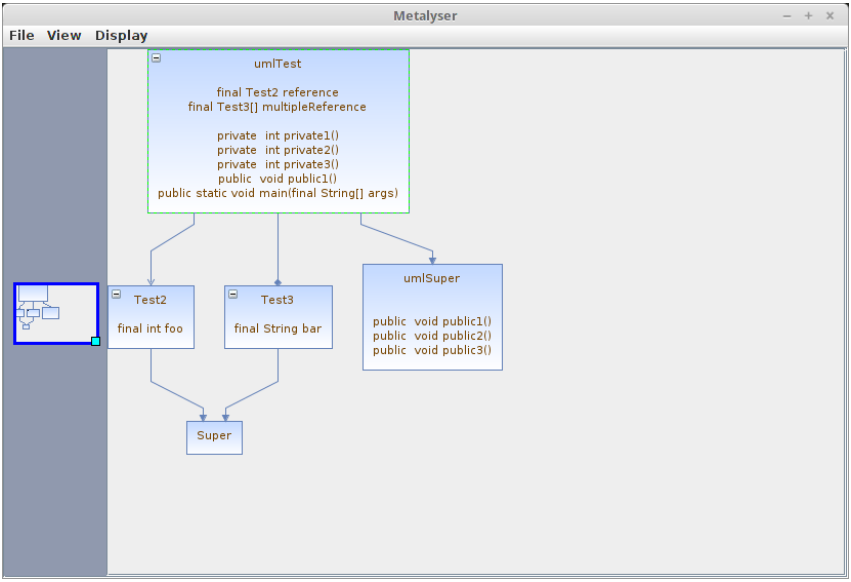

Рисунок 5.5. Диаграмма классов тестовой программы

Для класса `UmlTest` были получены следующие метрики:

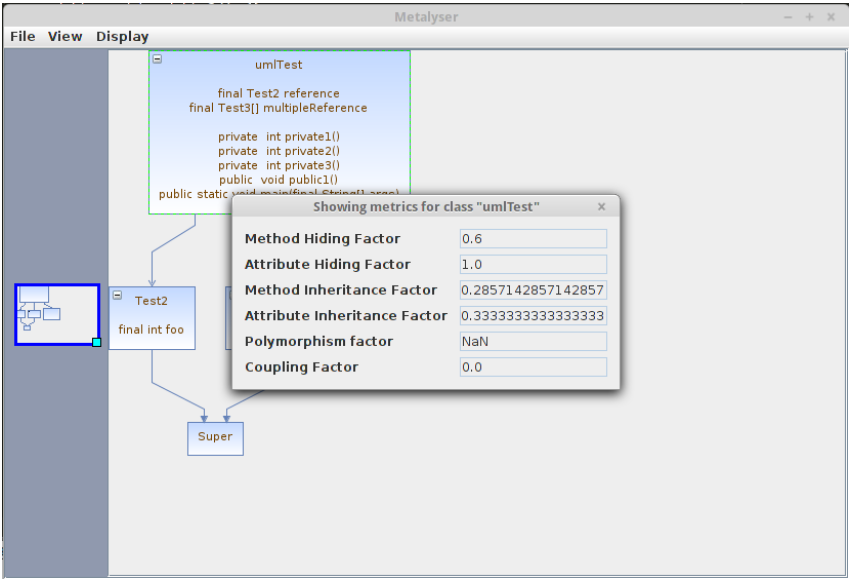

Рисунок 5.6. Набор метрик Абреу для класса UmlTest

5.2. Вывод

Тестирование разработанной программной системы показало ее соответствие сформулированным требованиям. Инструментальная среда позволяет извлекать и визуализировать несколько моделей программ (CFG и AST), а также строить UML-диаграммы классов и рассчитывать метрики Абреу для конкретного класса.

ЗАКЛЮЧЕНИЕ

В результате выполнения диссертации была разработана инструментальная среда, позволяющая автоматизировать процедуры анализа и верификации. Для решения данной задачи был предложен подход с использованием языконезависимой метамодели.

В ходе работы были рассмотрены методы повышения качества и используемые в них модели ПО (раздел 1). После обзора способов абстрагирования от языка программирования для унификации процедур анализа был проведен обзор существующих средств, использующих метамоделирование. На основе этого обзора было принято решение о целесообразности разработки собственного средства.

На основе анализа существующих средств и требований, поставленных в разделе 2, была предложена архитектура инструментальной среды, которая была разделена на три фрагмента - преобразователи, метамодель и процедуры анализа (раздел 3). Для каждой составляющей были предложены варианты реализации.

В разделе 4 была описана реализация разработанной архитектуры. При реализации использовались приемы и паттерны объектно-ориентированного программирования, использовался широкий круг библиотек для решения задач синтаксического разбора, сериализации и визуализации.

На заключительном этапе разработки (раздел 5) было проведено функциональное тестирование среды, показавшее соответствие разработанной системы поставленным требованиям.

Разработанную инструментальную среду можно применять для визуализации свойств программных систем при проведении различных неформальных методов повышения качества, например, аудита. Функция подсчета метрик позволяет оценивать характеристики проекта на поздних этапах его жизненного цикла, а визуализацию графа потока управления можно применять для отладки каких-либо процедур анализируемой системы.

Библиотека с метамоделью может быть использована как промежуточное представление в формальных методах анализа. Так как метамодель является независимой от языка программирования, на котором написана анализируемая программная система, разработанные процедуры анализа можно применять для широкого набора систем.

Дальнейшим развитием проекта является разработка новых ви-

дов визуализации, процедур извлечения новых видов моделей и различных процедур анализа на основе этих моделей. Также планируется добавление преобразователей для других популярных языков программирования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. for Standardization International Organization. ISO 8402: 1994: Quality Management and Quality Assurance - Vocabulary. — International Organization for Standardization, 1994.
2. М.И. Глухих, В.М. Ицыксон. Программная инженерия. Обеспечение качества программных средств методами статического анализа. — Санкт Петербург : Издательство Политехнического университета, 2011.
3. Ковалёв С. П. Применение формальных методов для обеспечения качества вычислительных систем // Вестник Новосибирского государственного университета. — 2004. — Т. IV, № 2. — С. 49–74.
4. В.В. Кулямин. Методы верификации программного обеспечения. — Институт системного программирования РАН, 2008.
5. Automation of GUI testing using a model-driven approach / Marlon Vieira, Johanne Leduc, Bill Hasling et al. // AST '06: Proceedings of the 2006 international workshop on Automation of software test. — New York, NY, USA : ACM, 2006. — P. 9–14.
6. Barnett Mike, Schulte Wolfram. Spying on Components: A Runtime Verification Technique // Proc. of the Workshop on Specification and Verification of Component- Based Systems OOPSLA 2001. — 2001.
7. Piefel M. A Common Metamodel for Code Generation // Proceedings of the 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications. — 2006.
8. Nierstrasz Oscar, Ducasse Stéphane, Gîrba Tudor. The story of moose: an agile reengineering environment. — ACM, 2005. — P. 1–10.
9. A Programming Language Independent Framework for Metrics-based Software Evolution and Analysis / Črt Gerlec, Gordana Rakić, Zoran Budimac, Marjan Heričko // Computer Science and Information Systems. — 2012. — Sep. — Vol. 9, no. 3. — P. 1155–1186.
10. Lattner Chris, Adve Vikram. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). — Palo Alto, California, 2004. — Mar.
11. Fowler Martin. UML Distilled: A Brief Guide to the Standard Object

Modeling Language. — 3 edition. — Boston, MA : Addison-Wesley, 2003. — ISBN: 978-0-321-19368-1.
12. Meta Object Facility (MOF) 2.0 Core Specification. — 2003. — Version 2.
13. MDA Guide Version 1.0.1 : Rep. / Object Management Group (OMG) ; Executor: J. Miller, J. Mukerji : 2003.
14. Overbeek J.F. Meta Object Facility (MOF): investigation of the state of the art. — 2006. — June.
15. Object Management Group (OMG). XML Meta-Data Interchange XMI 2.1.1. — formal/2007-12-01. — 2007.
16. Aho A., Sethi R., Ullman J. Compilers: Principles, Techniques and Tools. — Addison-Wesley, 1986.
17. Design patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. — Pearson Education, 1994.
18. Maven Documentation : Rep. / Apache Software Foundation ; Executor: Apache Maven Project : Apache Maven Project, 2001 - 2005. — URL: `http://maven.apache.org/guides/index.html`.
19. The Java Language Specification, Java SE 7 Edition / James Gosling, Bill Joy, Guy Steele et al. — Addison Wesley, 2013.
20. 222 JSR. The Java Architecture for XML Binding (JAXB) 2.2. — 2009. — URL: `http://jcp.org/en/jsr/detail?id=222`.
21. Gallagher Niall. Simple XML Framework Projekt. — online. — 2011. — URL: `http://simple.sourceforge.net/`.
22. Parr Terence. The Definitive ANTLR Reference: Building Domain-Specific Languages. — Pragmatic Bookshelf, 2007. — P. 376. — ISBN: 0978739256.
23. Mahmoud Khaled Zuhair. Compiler Construction using Java, JavaCC and YaCC, Anthony J. Dos Reis. Wiley (January, 2012) // Computer Science Review. — 2013. — Vol. 10. — P. 31–34.
24. The MOOD2 Metrics Set : Rep. ; Executor: Fernando Brito e Abreu : 1998.

ПРИЛОЖЕНИЕ А
ПРИМЕР МЕТАМОДЕЛИ, СЕРИАЛИЗОВАННОЙ В XML

Листинг А.1. Исходная программа

```
class Condition {
    public static void main(String[] args) {
        boolean learning = true;

        if (learning) {
            System.out.println("Java programmer");
        } else {
            System.out.println("What are you doing here?");
        }
    }
}
```

Листинг А.2. Сериализованная метамодель

```
<metamodel>
 <units class="java.util.LinkedList">
  <translationUnit>
   <imports class="java.util.LinkedList"/>
   <types class="java.util.LinkedList">
    <declaration class="edu.diploma.metamodel.declarations.ClassDecl">
     <name>Condition</name>
     <visibility>DEFAULT</visibility>
     <modifiers class="java.util.LinkedList"/>
     <annotations class="java.util.LinkedList"/>
     <templates class="java.util.Collections$EmptyList"/>
     <inherits class="java.util.LinkedList"/>
     <body>
      <name></name>
      <visibility>DEFAULT</visibility>
      <modifiers class="java.util.LinkedList"/>
      <annotations class="java.util.LinkedList"/>
      <decls class="java.util.LinkedList">
       <declaration class="edu.diploma.metamodel.declarations.
           FunctionDecl">
        <name>main</name>
        <visibility>PUBLIC</visibility>
        <modifiers class="java.util.LinkedList">
         <string>static</string>
        </modifiers>
        <annotations class="java.util.LinkedList"/>
        <retType class="edu.diploma.metamodel.types.PrimitiveType">
         <name>void</name>
        </retType>
        <exceptions class="java.util.Collections$EmptyList"/>
        <params class="java.util.LinkedList">
         <parameterDecl>
```

```
32            <name>args</name>
33            <visibility>DEFAULT</visibility>
34            <modifiers class="java.util.LinkedList"/>
35            <annotations class="java.util.LinkedList"/>
36            <value>
37             <name>args</name>
38             <visibility>DEFAULT</visibility>
39             <modifiers class="java.util.LinkedList"/>
40             <annotations class="java.util.LinkedList"/>
41             <type class="edu.diploma.metamodel.types.ArrayType">
42              <type class="edu.diploma.metamodel.types.ClassType">
43               <name>String</name>
44               <templates class="java.util.Collections$EmptyList"/>
45              </type>
46             </type>
47            </value>
48            <variadic>false</variadic>
49           </parameterDecl>
50          </params>
51          <templates class="java.util.Collections$EmptyList"/>
52          <body>
53           <statements class="java.util.LinkedList">
54            <statement class="edu.diploma.metamodel.statements.
                  VariableDeclStatement">
55             <variable>
56              <name>learning</name>
57              <visibility>DEFAULT</visibility>
58              <modifiers class="java.util.LinkedList"/>
59              <annotations class="java.util.LinkedList"/>
60              <type class="edu.diploma.metamodel.types.PrimitiveType">
61               <name>boolean</name>
62              </type>
63              <value class="edu.diploma.metamodel.literals.BooleanLiteral
                    ">
64               <type class="edu.diploma.metamodel.types.PrimitiveType">
65                <name>boolean</name>
66               </type>
67               <value>true</value>
68              </value>
69             </variable>
70            </statement>
71            <statement class="edu.diploma.metamodel.statements.
                  IfStatement">
72             <condition class="edu.diploma.metamodel.expressions.
                   VariableReference">
73              <type class="edu.diploma.metamodel.types.PrimitiveType">
74               <name>unknown</name>
75              </type>
76              <name>learning</name>
77             </condition>
78             <ifer class="edu.diploma.metamodel.statements.StatementBlock
                   ">
79              <statements class="java.util.LinkedList">
80               <statement class="edu.diploma.metamodel.expressions.
                     FunctionCall">
81                <type class="edu.diploma.metamodel.types.PrimitiveType">
82                 <name>unknown</name>
83                </type>
```

```
84                <name>println</name>
85                <caller class="edu.diploma.metamodel.expressions.
                      AttributeAccess">
86                 <type class="edu.diploma.metamodel.types.PrimitiveType">
87                  <name>unknown</name>
88                 </type>
89                 <expr class="edu.diploma.metamodel.expressions.
                       VariableReference">
90                  <type class="edu.diploma.metamodel.types.PrimitiveType"
                        >
91                   <name>unknown</name>
92                  </type>
93                  <name>System</name>
94                 </expr>
95                 <name>out</name>
96                </caller>
97                <params class="java.util.LinkedList">
98                 <expression class="edu.diploma.metamodel.literals.
                       StringLiteral">
99                  <type class="edu.diploma.metamodel.types.ClassType">
100                  <name>java.lang.String</name>
101                  <templates class="java.util.Collections$EmptyList"/>
102                 </type>
103                 <value>&amp;quot;Java programmer&amp;quot;</value>
104                </expression>
105               </params>
106               <templateParams class="java.util.Collections$EmptyList"/>
107              </statement>
108             </statements>
109            </ifer>
110            <elser class="edu.diploma.metamodel.statements.
                   StatementBlock">
111             <statements class="java.util.LinkedList">
112              <statement class="edu.diploma.metamodel.expressions.
                     FunctionCall">
113               <type class="edu.diploma.metamodel.types.PrimitiveType">
114                <name>unknown</name>
115               </type>
116               <name>println</name>
117               <caller class="edu.diploma.metamodel.expressions.
                      AttributeAccess">
118                <type class="edu.diploma.metamodel.types.PrimitiveType">
119                 <name>unknown</name>
120                </type>
121                <expr class="edu.diploma.metamodel.expressions.
                       VariableReference">
122                 <type class="edu.diploma.metamodel.types.PrimitiveType"
                        >
123                  <name>unknown</name>
124                 </type>
125                 <name>System</name>
126                </expr>
127                <name>out</name>
128               </caller>
129               <params class="java.util.LinkedList">
130                <expression class="edu.diploma.metamodel.literals.
                       StringLiteral">
131                 <type class="edu.diploma.metamodel.types.ClassType">
```

```
                    <name>java.lang.String</name>
                    <templates class="java.util.Collections$EmptyList"/>
                   </type>
                   <value>&quot;What are you doing here?&quot;</value>
                  </expression>
                 </params>
                 <templateParams class="java.util.Collections$EmptyList"/>
                </statement>
               </statements>
              </elser>
             </statement>
            </statements>
           </body>
          </declaration>
         </decls>
        </body>
       </declaration>
      </types>
     </translationUnit>
    </units>
</metamodel>
```

ПРИЛОЖЕНИЕ Б
ГРАММАТИКА ЯЗЫКА JAVA ДЛЯ ПОСТРОЕНИЯ МОДЕЛИ АНАЛИЗИРУЕМОЙ ПРОГРАММНОЙ СИСТЕМЫ

Листинг Б.1. Грамматика языка Java

```
grammar JavaGrammar;

compilationUnit
    :   packageDeclaration? importDeclaration* typeDeclaration* EOF
    ;

packageDeclaration
    :   annotation*  'package' qualifiedName ';'
    ;

importDeclaration
    :   'import' 'static'? qualifiedName ('.' '*')? ';'
    ;

typeDeclaration
    :   classOrInterfaceModifier*
        ( classDeclaration
        | enumDeclaration
        | interfaceDeclaration
        | annotationTypeDeclaration
        )
    |   ';'
    ;

modifier
    :   classOrInterfaceModifier
    |   (   t='native'
        |   t='synchronized'
        |   t='transient'
        |   t='volatile'
        )
    ;

classOrInterfaceModifier
    :   annotation
    |   (   t='public'
        |   t='protected'
        |   t='private'
        |   t='static'
        |   t='abstract'
        |   t='final'
        |   t='strictfp'
```

```
        )
    ;

variableModifier
    :   'final' | annotation
    ;

classDeclaration
    :   'class' Identifier typeParameters?
        ('extends' type)?
        ('implements' typeList)?
        classBody
    ;

typeParameters
    :   '<' typeParameter (',' typeParameter)* '>'
    ;

typeParameter
    :   Identifier ('extends' typeBound)?
    ;

typeBound
    :   type ('&' type)*
    ;

enumDeclaration
    :   ENUM Identifier ('implements' typeList)?
        '{' enumConstants? ','?
        enumBodyDeclarations? '}'
    ;

enumConstants
    :   enumConstant (',' enumConstant)*
    ;

enumConstant
    :   annotation* Identifier arguments? classBody?
    ;

enumBodyDeclarations
    :   ';' classBodyDeclaration*
    ;

interfaceDeclaration
    :   'interface' Identifier typeParameters?
        ('extends' typeList)? interfaceBody
    ;

typeList
    :   type (',' type)*
    ;

classBody
    :   '{' classBodyDeclaration* '}'
    ;

interfaceBody
```

```
101     :   '{' interfaceBodyDeclaration* '}'
102     ;
103
104 classBodyDeclaration
105     :   ';'
106     |   'static'? block
107     |   modifier* memberDeclaration
108     ;
109
110 memberDeclaration
111     :   methodDeclaration
112     |   genericMethodDeclaration
113     |   fieldDeclaration
114     |   constructorDeclaration
115     |   genericConstructorDeclaration
116     |   interfaceDeclaration
117     |   annotationTypeDeclaration
118     |   classDeclaration
119     |   enumDeclaration
120     ;
121
122 methodDeclaration
123     :   (type | 'void')
124         Identifier formalParameters ('[' ']')*
125         ('throws' qualifiedNameList)?
126         (methodBody | ';')
127     ;
128
129 genericMethodDeclaration
130     :   typeParameters methodDeclaration
131     ;
132
133 constructorDeclaration
134     :   Identifier formalParameters ('throws' qualifiedNameList)?
135         constructorBody
136     ;
137
138 genericConstructorDeclaration
139     :   typeParameters constructorDeclaration
140     ;
141
142 fieldDeclaration
143     :   type variableDeclarators ';'
144     ;
145
146 interfaceBodyDeclaration
147     :   modifier* interfaceMemberDeclaration
148     |   ';'
149     ;
150
151 interfaceMemberDeclaration
152     :   constDeclaration
153     |   interfaceMethodDeclaration
154     |   genericInterfaceMethodDeclaration
155     |   interfaceDeclaration
156     |   annotationTypeDeclaration
157     |   classDeclaration
158     |   enumDeclaration
```

```
159     ;
160
161 constDeclaration
162     :   type constantDeclarator (',' constantDeclarator)* ';'
163     ;
164
165 constantDeclarator
166     :   Identifier ('[' ']')* '=' variableInitializer
167     ;
168
169 interfaceMethodDeclaration
170     :   (type | 'void')
171         Identifier formalParameters ('[' ']')*
172         ('throws' qualifiedNameList)? ';'
173     ;
174
175 genericInterfaceMethodDeclaration
176     :   typeParameters interfaceMethodDeclaration
177     ;
178
179 variableDeclarators
180     :   variableDeclarator (',' variableDeclarator)*
181     ;
182
183 variableDeclarator
184     :   variableDeclaratorId ('=' variableInitializer)?
185     ;
186
187 variableDeclaratorId
188     :   Identifier ('[' ']')*
189     ;
190
191 variableInitializer
192     :   arrayInitializer | expression
193     ;
194
195 arrayInitializer
196     :   '{' (variableInitializer (',' variableInitializer)* (',')? )? '
            }'
197     ;
198
199 enumConstantName
200     :   Identifier
201     ;
202
203 type
204     :   classOrInterfaceType ('[' ']')*
205     |   primitiveType ('[' ']')*
206     ;
207
208 classOrInterfaceType
209     :   Identifier typeArguments? ( '.' Identifier typeArguments?)*
210     ;
211
212 primitiveType
213     :   'boolean'
214     |   'char'
215     |   'byte'
```

```
216     |   'short'
217     |   'int'
218     |   'long'
219     |   'float'
220     |   'double'
221     ;
222
223 typeArguments
224     :   '<' typeArgument (',' typeArgument)* '>'
225     ;
226
227 typeArgument
228     :   type
229     |   '?'
230     |   '?' 'extends' type
231     |   '?' 'super' type
232     ;
233
234 qualifiedNameList
235     :   qualifiedName (',' qualifiedName)*
236     ;
237
238 formalParameters
239     :   '(' formalParameterList? ')'
240     ;
241
242 formalParameterList
243     :   formalParameter (',' formalParameter)* (',' lastFormalParameter
            )?
244     |   lastFormalParameter
245     ;
246
247 formalParameter
248     :   variableModifier* type variableDeclaratorId
249     ;
250
251 lastFormalParameter
252     :   variableModifier* type '...' variableDeclaratorId
253     ;
254
255 methodBody
256     :   block
257     ;
258
259 constructorBody
260     :   block
261     ;
262
263 qualifiedName
264     :   Identifier ('.' Identifier)*
265     ;
266
267 literal
268     :   IntegerLiteral
269     |   FloatingPointLiteral
270     |   CharacterLiteral
271     |   StringLiteral
272     |   BooleanLiteral
```

```
273     |   'null'
274     ;
275
276 // ANNOTATIONS
277
278 annotation
279     :   '@' annotationName ( '(' ( elementValuePairs | elementValue )?
            ')' )?
280     ;
281
282 annotationName
283     : qualifiedName
284
285 elementValuePairs
286     :   elementValuePair (',' elementValuePair)*
287     ;
288
289 elementValuePair
290     :   Identifier '=' elementValue
291     ;
292
293 elementValue
294     :   expression
295     |   annotation
296     |   elementValueArrayInitializer
297     ;
298
299 elementValueArrayInitializer
300     :   '{' (elementValue (',' elementValue)*)? (',')? '}'
301     ;
302
303 annotationTypeDeclaration
304 returns [AnnotationDecl result]
305     :   '@' 'interface' Identifier annotationTypeBody
306     ;
307
308 annotationTypeBody
309     :   '{'
310         ( annotationTypeElementDeclaration*
311         | annotationConstantsDeclaration+
312         )
313         '}'
314     ;
315
316 annotationTypeElementDeclaration
317     :   modifier* annotationTypeElementRest
318     |   ';'
319     ;
320
321 annotationTypeElementRest
322     :   annotationMethod ';'
323     |   classDeclaration ';'?
324     |   interfaceDeclaration ';'?
325     |   enumDeclaration ';'?
326     |   annotationTypeDeclaration ';'?
327     ;
328
329
```

```
330 annotationConstantsDeclaration
331     :   modifier* annotationConstants
332     ;
333
334 annotationConstants
335     :   type annotationConstantRest
336     ;
337
338 annotationMethod
339     :   type annotationMethodRest
340     ;
341
342 annotationMethodRest
343     :   Identifier '(' ')' defaultValue?
344     ;
345
346 annotationConstantRest
347     :   variableDeclarators
348     ;
349
350 defaultValue
351     :   'default' elementValue
352     ;
353
354 // STATEMENTS / BLOCKS
355
356 block
357     :   '{' blockStatement* '}'
358     ;
359
360 blockStatement
361     :   localVariableDeclarationStatement
362     |   statement
363     |   typeDeclaration
364     ;
365
366 localVariableDeclarationStatement
367     :    localVariableDeclaration ';'
368     ;
369
370 localVariableDeclaration
371     :   variableModifier* type variableDeclarators
372     ;
373
374 statement
375     :   block
376     |   ASSERT expression (':' expression)? ';'
377     |   'if' parExpression statement ('else' statement)?
378     |   forStatement
379     |   forEach
380     |   'while' parExpression statement
381     |   'do' statement 'while' parExpression ';'
382     |   'try' block (catchClause+ finallyBlock? | finallyBlock)
383     |   'try' resourceSpecification block catchClause* finallyBlock?
384     |   'switch' parExpression '{'
385         switchBlockStatementGroup*
386         switchLabel* '}'
387     |   'synchronized' parExpression block
```

```
388     |   'return' expression? ';'
389     |   'throw' expression ';'
390     |   'break' Identifier? ';'
391     |   'continue' Identifier? ';'
392     |   ';'
393     |   statementExpression ';'
394     |   Identifier ':' statement
395     ;
396
397 catchClause
398     :   'catch' '(' variableModifier* catchType Identifier ')' block
399     ;
400
401 catchType
402     :   qualifiedName ('|' qualifiedName)*
403     ;
404
405 finallyBlock
406     :   'finally' block
407     ;
408
409 resourceSpecification
410     :   '(' resources ';'? ')'
411     ;
412
413 resources
414     :   resource (';' resource)*
415     ;
416
417 resource
418     :   variableModifier* classOrInterfaceType
419         variableDeclaratorId '=' expression
420     ;
421
422 switchBlockStatementGroup
423     :   switchLabel+ blockStatement+
424     ;
425
426 switchLabel
427     :   'case' constantExpression ':'
428     |   'case' enumConstantName ':'
429     |   'default' ':'
430     ;
431
432 forStatement
433     :   'for' '(' forInit? ';' expression? ';' forUpdate? ')' statement
434     ;
435
436 forEach
437     :   'for' '(' enhancedForControl ')' statement
438     ;
439
440 forInit
441     :   localVariableDeclaration
442     |   expressionList
443     ;
444
445 enhancedForControl
```

```
446     :   variableModifier* type Identifier ':' expression
447     ;
448
449 forUpdate
450     :   expressionList
451     ;
452
453 // EXPRESSIONS
454
455 parExpression
456     :   '(' expression ')'
457     ;
458
459 expressionList
460     :   expression (',' expression)*
461     ;
462
463 statementExpression
464     :   expression
465     ;
466
467 constantExpression
468     :   expression
469     ;
470
471 expression
472     :   primary
473     |   expression '.' Identifier
474     |   expression '.' 'this'
475     |   expression '.' 'new' nonWildcardTypeArguments? innerCreator
476     |   expression '.' 'super' superSuffix
477     |   expression '.' explicitGenericInvocation
478     |   expression '[' expression ']'
479     |   Identifier '(' expressionList? ')'
480     |   'this' '(' expressionList? ')'
481     |   expression '.' Identifier '(' expressionList? ')'
482     |   'new' creator
483     |   '(' type ')' expression
484     |   expression ('++' | '--')
485     |   ('+'| '-' | '++' | '--') expression
486     |   ('~' | '!') expression
487     |   expression ('*'| '/' | '%') expression
488     |   expression ('+' | '-') expression
489     |   expression ('<' '<' | '>' '>' '>' | '>' '>') expression
490     |   expression ('<=' | '>=' | '>' | '<') expression
491     |   expression 'instanceof' type
492     |   expression ('==' | '!=') expression
493     |   expression '&' expression
494     |   expression '^' expression
495     |   expression '|' expression
496     |   expression '&&' expression
497     |   expression '||' expression
498     |   cond=expression '?' ifBranch=expression ':' elseBranch=
            expression
499     |   <assoc=right> expression '=' expression
500     |   <assoc=right> expression
501         (   '+='
502         |   '-='
```

```
503         |   '*='
504         |   '/='
505         |   '&='
506         |   '|='
507         |   '^='
508         |   '>>='
509         |   '>>>='
510         |   '<<='
511         |   '%='
512         )
513         expression
514     ;
515
516 primary
517     :   '(' expression ')'
518     |   ('this' | 'super')
519     |   literal
520     |   Identifier
521     |   type '.' 'class'
522     |   'void' '.' 'class'
523     |   nonWildcardTypeArguments
524         (explicitGenericInvocationSuffix | 'this' arguments)
525     ;
526
527 creator
528     :   nonWildcardTypeArguments? createdName classCreatorRest
529     |   createdName arrayCreatorRest
530     ;
531
532 createdName
533     :   Identifier typeArgumentsOrDiamond?
534         ( '.' Identifier typeArgumentsOrDiamond?)*
535     |   primitiveType
536     ;
537
538 innerCreator
539     :   Identifier nonWildcardTypeArgumentsOrDiamond? classCreatorRest
540     ;
541
542 arrayCreatorRest
543     :   '[' ']' ('[' ']')*
544         arrayInitializer
545     |   '[' expression ']' ('[' expression ']')* ('[' ']')*
546     ;
547
548 classCreatorRest
549     :   arguments classBody?
550     ;
551
552 explicitGenericInvocation
553     :   nonWildcardTypeArguments explicitGenericInvocationSuffix
554     ;
555
556 nonWildcardTypeArguments
557     :   '<' typeList '>'
558     ;
559
560 typeArgumentsOrDiamond
```

```
    :   '<' '>'
    |   typeArguments
    ;

nonWildcardTypeArgumentsOrDiamond
    :   '<' '>'
    |   nonWildcardTypeArguments
    ;

superSuffix
    :   arguments
    |   '.' Identifier arguments?
    ;

explicitGenericInvocationSuffix
returns [String name, List<Expression> args]
    :   'super' superSuffix
    |   Identifier arguments
    ;

arguments
    :   '(' expressionList? ')'
    ;
```

ПРИЛОЖЕНИЕ В
ГРАММАТИКА ЯЗЫКА С ДЛЯ ПОСТРОЕНИЯ МОДЕЛИ АНАЛИЗИРУЕМОЙ ПРОГРАММНОЙ СИСТЕМЫ

Листинг В.1. Грамматика языка С

```
1  grammar C;
2
3  primaryExpression
4      :   Identifier
5      |   Constant
6      |   StringLiteral+
7      |   '(' expression ')'
8      ;
9
10 postfixExpression
11     :   primaryExpression
12     |   postfixExpression '[' expression ']'
13     |   postfixExpression '(' argumentExpressionList? ')'
14     |   postfixExpression '.' Identifier
15     |   postfixExpression '->' Identifier
16     |   postfixExpression '++'
17     |   postfixExpression '--'
18     |   '(' typeName ')' '{' initializerList '}'
19     |   '(' typeName ')' '{' initializerList ',' '}'
20     ;
21
22 argumentExpressionList
23     :   assignmentExpression
24     |   argumentExpressionList ',' assignmentExpression
25     ;
26
27 unaryExpression
28     :   postfixExpression
29     |   '++' unaryExpression
30     |   '--' unaryExpression
31     |   unaryOperator castExpression
32     |   'sizeof' unaryExpression
33     |   'sizeof' '(' typeName ')'
34     ;
35
36 unaryOperator
37     :   '&' | '*' | '+' | '-' | '~' | '!'
38     ;
39
40 castExpression
41     :   unaryExpression
42     |   '(' typeName ')' castExpression
```

```
43      ;
44
45  multiplicativeExpression
46      :   castExpression
47      |   multiplicativeExpression '*' castExpression
48      |   multiplicativeExpression '/' castExpression
49      |   multiplicativeExpression '%' castExpression
50      ;
51
52  additiveExpression
53      :   multiplicativeExpression
54      |   additiveExpression '+' multiplicativeExpression
55      |   additiveExpression '-' multiplicativeExpression
56      ;
57
58  shiftExpression
59      :   additiveExpression
60      |   shiftExpression '<<' additiveExpression
61      |   shiftExpression '>>' additiveExpression
62      ;
63
64  relationalExpression
65      :   shiftExpression
66      |   relationalExpression '<' shiftExpression
67      |   relationalExpression '>' shiftExpression
68      |   relationalExpression '<=' shiftExpression
69      |   relationalExpression '>=' shiftExpression
70      ;
71
72  equalityExpression
73      :   relationalExpression
74      |   equalityExpression '==' relationalExpression
75      |   equalityExpression '!=' relationalExpression
76      ;
77
78  andExpression
79      :   equalityExpression
80      |   andExpression '&' equalityExpression
81      ;
82
83  exclusiveOrExpression
84      :   andExpression
85      |   exclusiveOrExpression '^' andExpression
86      ;
87
88  inclusiveOrExpression
89      :   exclusiveOrExpression
90      |   inclusiveOrExpression '|' exclusiveOrExpression
91      ;
92
93  logicalAndExpression
94      :   inclusiveOrExpression
95      |   logicalAndExpression '&&' inclusiveOrExpression
96      ;
97
98  logicalOrExpression
99      :   logicalAndExpression
100     |   logicalOrExpression '||' logicalAndExpression
```

```
    ;

conditionalExpression
    :   logicalOrExpression ('?' expression ':' conditionalExpression)?
    ;

assignmentExpression
    :   conditionalExpression
    |   unaryExpression assignmentOperator assignmentExpression
    ;

assignmentOperator
    :   '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '&=' |
        '^=' | '|='
    ;

expression
    :   assignmentExpression
    |   expression ',' assignmentExpression
    ;

constantExpression
    :   conditionalExpression
    ;

declaration
    :   declarationSpecifiers initDeclaratorList? ';'
    |   staticAssertDeclaration
    ;

declarationSpecifiers
    :   declarationSpecifier+
    ;

declarationSpecifier
    :   storageClassSpecifier
    |   typeSpecifier
    |   typeQualifier
    |   alignmentSpecifier
    ;

initDeclaratorList
    :   initDeclarator
    |   initDeclaratorList ',' initDeclarator
    ;

initDeclarator
    :   declarator
    |   declarator '=' initializer
    ;

storageClassSpecifier
    :   'typedef'
    |   'extern'
    |   'static'
    |   'auto'
    |   'register'
    ;
```

```
158
159 typeSpecifier
160     :   ('void'
161     |   'char'
162     |   'short'
163     |   'int'
164     |   'long'
165     |   'float'
166     |   'double'
167     |   'signed'
168     |   'unsigned'
169     |   structOrUnionSpecifier
170     |   enumSpecifier
171     |   typedefName
172     ;
173
174 structOrUnionSpecifier
175     :   structOrUnion Identifier? '{' structDeclarationList '}'
176     |   structOrUnion Identifier
177     ;
178
179 structOrUnion
180     :   'struct'
181     |   'union'
182     ;
183
184 structDeclarationList
185     :   structDeclaration
186     |   structDeclarationList structDeclaration
187     ;
188
189 structDeclaration
190     :   specifierQualifierList structDeclaratorList? ';'
191     |   staticAssertDeclaration
192     ;
193
194 specifierQualifierList
195     :   typeSpecifier specifierQualifierList?
196     |   typeQualifier specifierQualifierList?
197     ;
198
199 structDeclaratorList
200     :   structDeclarator
201     |   structDeclaratorList ',' structDeclarator
202     ;
203
204 structDeclarator
205     :   declarator
206     |   declarator? ':' constantExpression
207     ;
208
209 enumSpecifier
210     :   'enum' Identifier? '{' enumeratorList '}'
211     |   'enum' Identifier? '{' enumeratorList ',' '}'
212     |   'enum' Identifier
213     ;
214
215 enumeratorList
```

```
216     :   enumerator
217     |   enumeratorList ',' enumerator
218     ;
219
220 enumerator
221     :   enumerationConstant
222     |   enumerationConstant '=' constantExpression
223     ;
224
225 enumerationConstant
226     :   Identifier
227     ;
228
229 typeQualifier
230     :   'const'
231     |   'restrict'
232     |   'volatile'
233     ;
234
235 declarator
236     :   pointer? directDeclarator gccDeclaratorExtension*
237     ;
238
239 directDeclarator
240     :   Identifier
241     |   '(' declarator ')'
242     |   directDeclarator '[' typeQualifierList? assignmentExpression? '
            ]'
243     |   directDeclarator '[' 'static' typeQualifierList?
             assignmentExpression ']'
244     |   directDeclarator '[' typeQualifierList 'static'
             assignmentExpression ']'
245     |   directDeclarator '[' typeQualifierList? '*' ']'
246     |   directDeclarator '(' parameterTypeList ')'
247     |   directDeclarator '(' identifierList? ')'
248     ;
249
250 nestedParenthesesBlock
251     :   (   ~('(' | ')')
252         |   '(' nestedParenthesesBlock ')'
253         )*
254     ;
255
256 pointer
257     :   '*' typeQualifierList?
258     |   '*' typeQualifierList? pointer
259     ;
260
261 typeQualifierList
262     :   typeQualifier
263     |   typeQualifierList typeQualifier
264     ;
265
266 parameterTypeList
267     :   parameterList
268     |   parameterList ',' '...'
269     ;
270
```

```
parameterList
    :   parameterDeclaration
    |   parameterList ',' parameterDeclaration
    ;

parameterDeclaration
    :   declarationSpecifiers declarator
    |   declarationSpecifiers abstractDeclarator?
    ;

identifierList
    :   Identifier
    |   identifierList ',' Identifier
    ;

typeName
    :   specifierQualifierList abstractDeclarator?
    ;

abstractDeclarator
    :   pointer
    |   pointer? directAbstractDeclarator gccDeclaratorExtension*
    ;

directAbstractDeclarator
    :   '[' typeQualifierList? assignmentExpression? ']'
    |   '[' '*' ']'
    |   '(' parameterTypeList? ')' gccDeclaratorExtension*
    |   directAbstractDeclarator '[' typeQualifierList?
        assignmentExpression? ']'
    |   directAbstractDeclarator '[' 'static' typeQualifierList?
        assignmentExpression ']'
    |   directAbstractDeclarator '[' typeQualifierList 'static'
        assignmentExpression ']'
    |   directAbstractDeclarator '[' '*' ']'
    |   directAbstractDeclarator '(' parameterTypeList? ')'
        gccDeclaratorExtension*
    ;

typedefName
    :   Identifier
    ;

initializer
    :   assignmentExpression
    |   '{' initializerList '}'
    |   '{' initializerList ',' '}'
    ;

initializerList
    :   designation? initializer
    |   initializerList ',' designation? initializer
    ;

designation
    :   designatorList '='
    ;
```

```
designatorList
    :   designator
    |   designatorList designator
    ;

designator
    :   '[' constantExpression ']'
    |   '.' Identifier
    ;

statement
    :   labeledStatement
    |   compoundStatement
    |   expressionStatement
    |   selectionStatement
    |   iterationStatement
    |   jumpStatement
    ;

labeledStatement
    :   Identifier ':' statement
    |   'case' constantExpression ':' statement
    |   'default' ':' statement
    ;

compoundStatement
    :   '{' blockItemList? '}'
    ;

blockItemList
    :   blockItem
    |   blockItemList blockItem
    ;

blockItem
    :   declaration
    |   statement
    ;

expressionStatement
    :   expression? ';'
    ;

selectionStatement
    :   'if' '(' expression ')' statement ('else' statement)?
    |   'switch' '(' expression ')' statement
    ;

iterationStatement
    :   'while' '(' expression ')' statement
    |   'do' statement 'while' '(' expression ')' ';'
    |   'for' '(' expression? ';' expression? ';' expression? ')'
        statement
    |   'for' '(' declaration expression? ';' expression? ')' statement
    ;

jumpStatement
    :   'goto' Identifier ';'
```

```
382     |   'continue' ';'
383     |   'break' ';'
384     |   'return' expression? ';'
385     ;
386
387 compilationUnit
388     :   translationUnit? EOF
389     ;
390
391 translationUnit
392     :   externalDeclaration
393     |   translationUnit externalDeclaration
394     ;
395
396 externalDeclaration
397     :   functionDefinition
398     |   declaration
399     |   ';' // stray ;
400     ;
401
402 functionDefinition
403     :   declarationSpecifiers? declarator declarationList?
            compoundStatement
404     ;
405
406 declarationList
407     :   declaration
408     |   declarationList declaration
409     ;
```