Todo list

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**

230100 – Информатика и вычислительная техника
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, 42 стр., 14 рис.

ABSTRACT

Report, 42 pages, 14 figures

СОДЕРЖАНИЕ

СПИСОК ОБОЗНАЧЕНИЙ И СОКРАЩЕНИЙ

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
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
SDL	Specification and Description Language, язык для описания моделей программных систем
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
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 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 1.3)
- Построение различных метрик программного кода
- Реинжиниринг программного обеспечения (оптимизация, рефакторинг и т.п.)
- Визуализация свойств программной системы
- и т.п.

Более подробно методы обеспечения качества рассмотрены в подразделе 1.1.

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 представлений программы, не зависящих от языка написания ее исходного кода, называемых метамоделями.

1 АНАЛИЗ ПОДХОДОВ И СРЕДСТВ ИНСТРУМЕНТИРОВАНИЯ ПРОГРАММ

1.1 Методы повышения качества

Существует две группы подходов по обеспечению качества программного обеспечения [1]:

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
- генераторы шаблонов программ
- генераторы программ
- 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, используется 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.

1.2 Классификация методов обеспечения качества

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

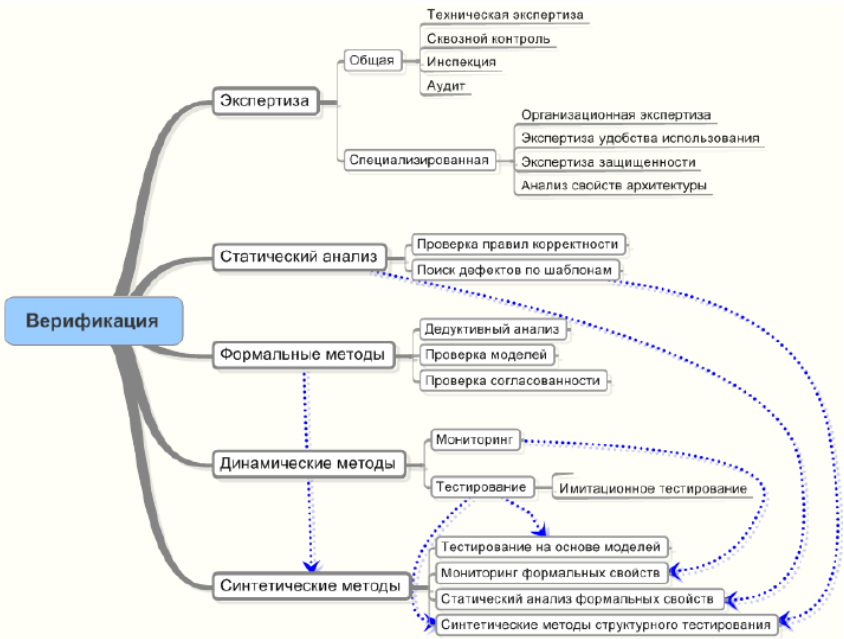

Рисунок 1.1. Схема используемой классификации методов верификации

Экспертиза позволяет находить ошибки, используя различные артефакты жизненного цикла системы, в отличие от формальных и динамических методов, и позволяет находить большое множество разновидностей ошибок. К ее недостаткам можно отнести невозможность автоматизации.

Статический анализ - это процесс выявления ошибок и недочетов в исходном коде программ. От остальных методов верификации его отделяет то, что статический анализ 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.

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.

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. Также данные методы позволяют найти только те ошибки в ПО, которые проявляются в его работе.

Синтетические методы объединяют в себе элементы некоторых способов повышения качества, описанных выше. Например, существуют динамические методы, использующие элементы формальных - тестирование на основе моделей (model driven testing) [4] и мониторинг формальных свойств (runtime verification) [5]. Цель таких

методов - объединить преимущества уже используемых подходов.

1.3 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 могу различаться по структуре и сложности и обладать различными свойствами. Существуют следующие виды моделей [1]:

- 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.

Рассказать про метамодели и их положение в иерархии моделирования

1.4 Анализ существующих решений

Обощенное решение задач анализа и верификации основывается на использовании метамоделей [6] - описаний модели системы, имеющих различные уровни детализации в зависимости от преследуемых целей.

На данный момент существует лишь небольшое количество инструментов, использующих метамоделирование. Рассмотрим их подробнее.

Moose

Moose является платформой для анализа программ и поддерживает большое количество различных видов анализа [7]:

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 1.2

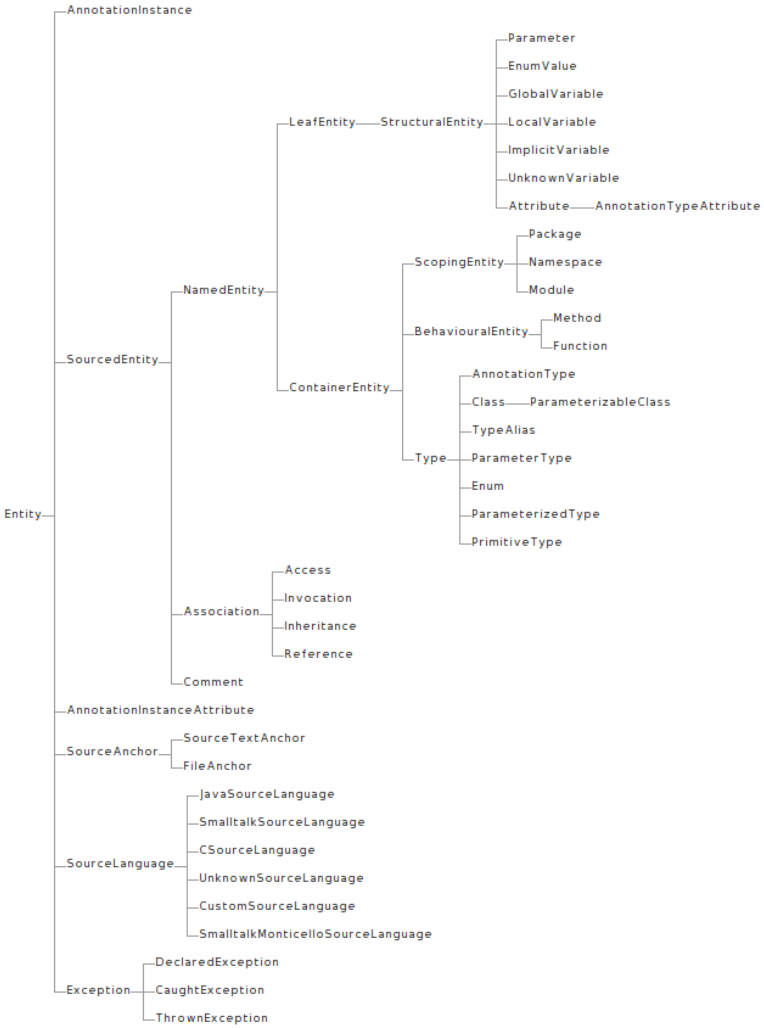

Рисунок 1.2. Структура метамоделей семейства FAMIX

Анализ программы происходит следующим образом:

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

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

3. Применение заданных алгоритмов анализа.

Архитектура средства приведена на рис 1.3

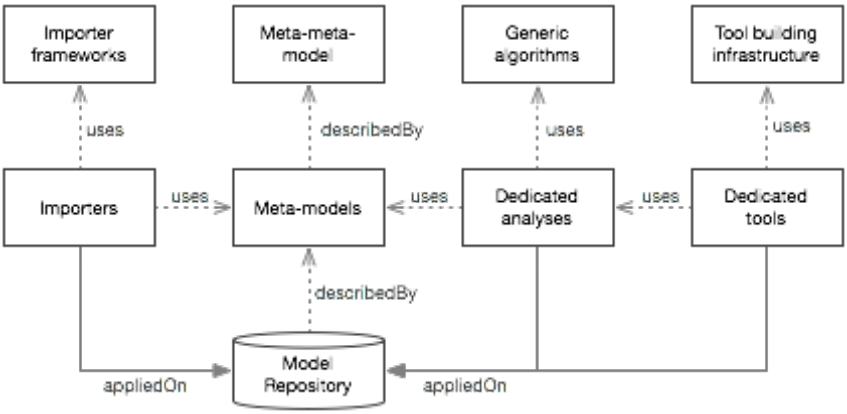

Рисунок 1.3. Архитектура фреймворка Moose

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.

SMIILE

SMIILE - среда, предназначенная для вычисления метрик анализируемой системы [8].

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 дерево разбора независимым от входного языка программирования.

На рис 1.4 изображена архитектура данного средства:

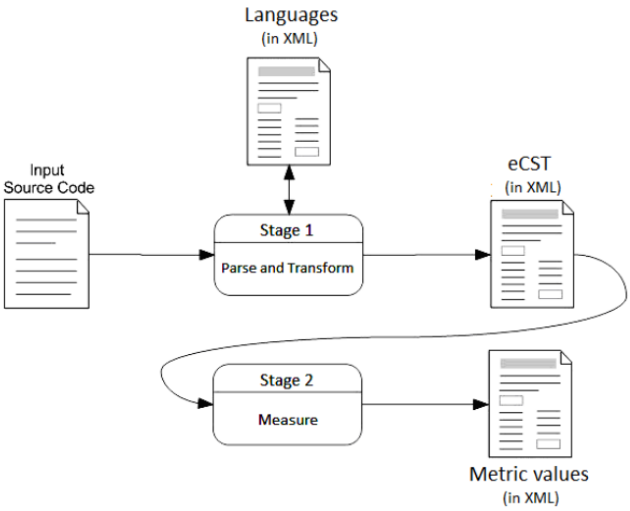

Рисунок 1.4. Архитектура SMIILE

Анализ программы происходит в две фазы:

1. Фаза 1
   - Определение языка программирования, на котором написана анализируемая программа.
   - Вызов парсера этого языка для построения CST и преобразование в eCST.
   - Вывод результата в формате XML.
2. Фаза 2

- Считывание eCST из файла.
- Подсчет метрик.
- Сохранение результата в формате XML.

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

LLVM

LLVM - фреймворк для анализа и трансформации программ, путем предоставления информации для трансформаций компилятору во время компиляции, линковки и исполнения [9].

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:

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

Архитектура LLVM приведена на рис 1.5:

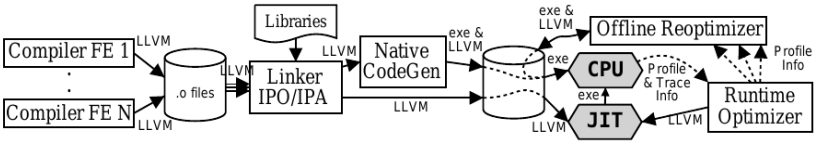

Рисунок 1.5. Архитектура LLVM

Front-end компиляторы транслируют исходную программу в промежуточное представление LLVM, которое затем компонуется LLVM-линкером. 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.

ULF-Ware

ULF-Ware является средством для генерации кода из модели, созданной при помощи языка SDL (Specification and Description Language) [6]. Данное средство использует метамодель под названием CeeJay, которая предназначена для описания систем на языках Java и C++.

Архитектура ULF-Ware приведена на рис 1.6:

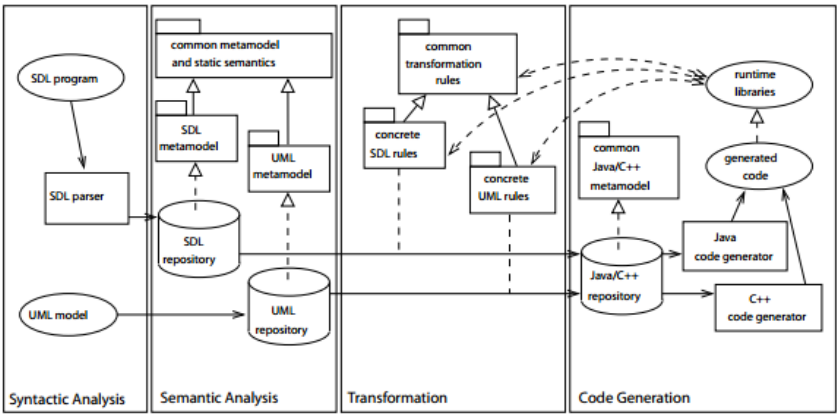

Рисунок 1.6. Архитектура ULF-Ware

Генерация кода происходит следующим образом: компилятор SDL создает модель программы, полученную из SDL-описания, 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++.

2 ПОСТАНОВКА ЗАДАЧИ

Как было сказано в разделе 1.3, 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 Постановка 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.

Разрабатываемое средство должно предоставлять следующие возможности:

1. Извлечение моделей, необходимых для написания процедур статического анализа и верификации
2. Визуализация следующих моделей и свойств:
    - CFG
    - AST
    - Диаграмма классов

- Подсчет метрик

Ядром всей системы и средством абстракции является метамодель, к ней предъявляются следующие требования:

1. Независимость от языка описания системы
2. Расширяемость - возможность добавлять новые сущности по мере необходимости
3. Простота в использовании

2.2 Выбор пути решения

Как видно из обзора в п. 1.4, 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
4. Анализ полученных результатов

2.3 Вывод

В результате успешного выполнения задач, приведенных в пп.2.2, необходимо получить инструментальную среду, поддерживающую несколько видов визуализации свойств программных систем, оставаясь при этот независимой от языка описания этих систем.

3 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ИНСТРУМЕНТАЛЬНОЙ СРЕДЫ

3.1 Структура программной системы

На основе анализа поставленных требований, а также уже существующих решений, была предложена следующая архитектура программной системы:

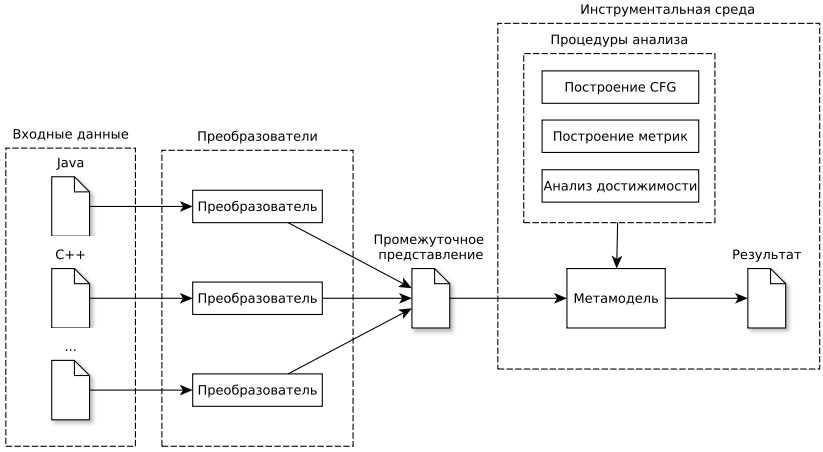

Рисунок 3.1. Архитектура программной системы

Как видно из рис. 3.1 система состоит из 3-х 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 с промежуточным представлением, то 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 к проектированию.

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 [10].

Для описания конструкций языка UML был разработан фреймворк MOF [11]. В дальнейшем обе эти концепции вошли в подход, называемый Model Driven Architecture (MDA) [12]. 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 [11].

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 так же называются *слоями спецификации языков* [13]. Каждый слой является уровнем абстракции - чем ниже слой, тем конкретнее описывается система.

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

Ниже приведен пример иерархии для конкретной системы:

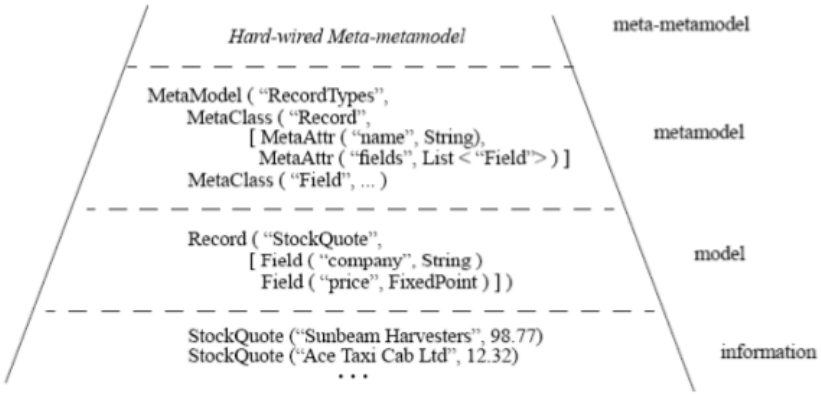

Рисунок 3.2. Пример иерархии моделей

Теоретически, иерархию можно дополнить дополнительными слоями над слоем M3, однако 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 так же объединять в пакеты, 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:

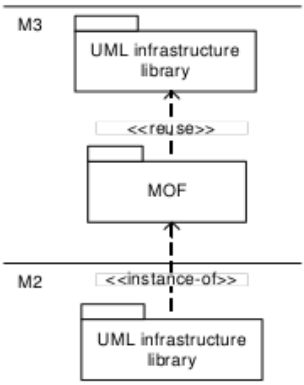

Рисунок 3.3. Взаимосвязь между пакетами MOF верхнего уровня

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

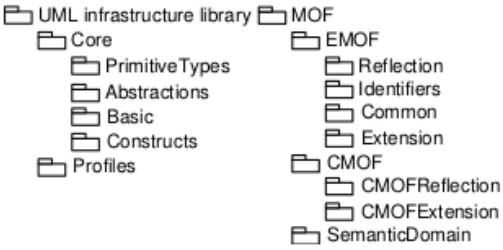

Рисунок 3.4. Дерево пакетов MOF

Опишем некоторые из приведенных пакетов:

Abstractions

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

Basic

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

Constructs

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

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:

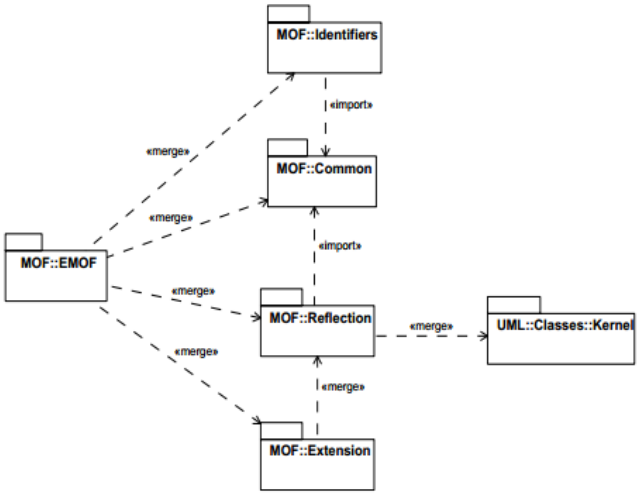

Рисунок 3.5. Структура EMOF

CMOF

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

3.2.4 Сериализация метамодели

Для сериализации UML моделей и их передачи между различными средствами OGM был разработан формат XMI. Но так как UML является совместимой со стандартом MOF, XMI может быть использован для сериализации любой MOF-совместимой метамодели [14].

В стандарт XMI входят следущие 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. 3.6:

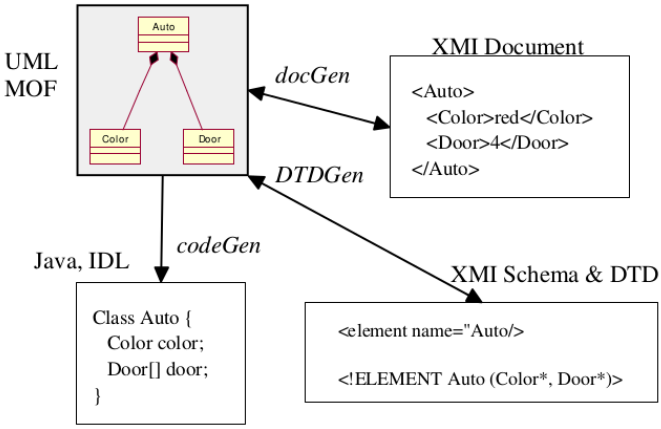

Рисунок 3.6. Отношение между MOF и XMI

Приведем пример сериализованной UML модели в формате XMI:

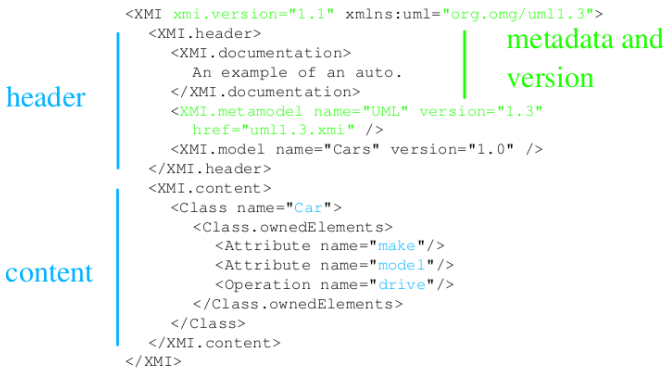

Рисунок 3.7. Пример сериализованной UML-модели

Таким образом, выбранный формат данных естественным образом ложится на разработанную архитектуру, а его стандартизированность облегчает взаимодействие между инструментальной средой и сторонними преобразователями. 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 [15]:

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 пользователю будет необходимо проверять корректность анализируемой системы сторонними средствами (например, при помощи компилятора языка программирования, на котором написана анализируемая система).

Получившаяся структура преобразователя изображена на рис. 3.8.

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

1. Написание кода лексера и парсера вручную (например, парсеров, использующие 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 лексера и парсера. К недостаткам данного метода можно отнести высокую трудоемкость и сложность написания.

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.

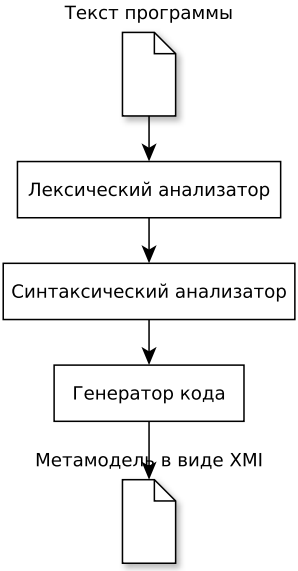

Рисунок 3.8. Архитектура преобразователя

На основе анализа достоинств и недостатков обоих методов, было принято решение использование автоматической генерации кода парсеров и лексеров разрабатываемых преобразователей. В случае нехватки производительности или объема предоставляемой памяти по результатам профилирование 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 рационально использование различных фреймворков для преобразования объектов программы в XML.

3.4 Проектирование графического интерфейса

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 паттерн "Посетитель"(Visitor) [16]. Применения данного паттерна позволит абстрагировать метамодель от алгоритмов над ней, выделив обход ее структуры в отдельный класс.

Необходимые виды визуализации, описанные в п. 2, имеют вид дерева или 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

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Глухих М.И., Ицыксон В.М. Программная инженерия. Обеспечение качества программных средств методами статического анализа. — Санкт Петербург : Издательство Политехнического университета, 2011.
2. Кулямин В.В. Методы верификации программного обеспечения. — Институт системного программирования РАН, 2008.
3. С. П. Ковалёв. Применение формальных методов для обеспечения качества вычислительных систем // Вестник Новосибирского государственного университета. — 2004. — Т. IV, № 2. — С. 49–74.
4. 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.
5. 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.
6. Piefel M. A Common Metamodel for Code Generation // Proceedings of the 3rd International Conference on Cybernetics and Information Technologies, Systems and Applications. — 2006.
7. Nierstrasz Oscar, Ducasse Stéphane, Gîrba Tudor. The story of moose: an agile reengineering environment. — ACM, 2005. — P. 1–10.
8. 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.
9. 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.
10. 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.
11. Meta Object Facility (MOF) 2.0 Core Specification. — 2003. — Ver-

sion 2.
12. MDA Guide Version 1.0.1 : Rep. / Object Management Group (OMG) ; Executor: J. Miller, J. Mukerji : 2003.
13. Overbeek J.F. Meta Object Facility (MOF): investigation of the state of the art. — 2006. — June.
14. Object Management Group (OMG). XML Meta-Data Interchange XMI 2.1.1. — formal/2007-12-01. — 2007.
15. Aho A., Sethi R., Ullman J. Compilers: Principles, Techniques and Tools. — Addison-Wesley, 1986.
16. Design patterns: elements of reusable object-oriented software / Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. — Pearson Education, 1994.