Санкт-Петербургский государственный политехнический университет
Кафедра компьютерных систем и программных технологий

РЕФЕРАТ

Дисциплина: **Современные проблемы информатики и вычислительной техники**

Тема: **Анализ подходов и средств для верификации программ. Использование фреймворков как средств автоматизации**

Выполнил студент гр. 63501/13 А.М. Половцев
Преподаватель В.Ф. Мелехин
“____”_____ 2013 г.

Санкт-Петербург
2013

Содержание

1 Введение

С развитием вычислительных систем и ростом в них доли программной составляющей, сложность разрабатываемых программ постоянно возрастает. Также, вследствие большой конкуренции на рынке программного обеспечения, разработчики вынуждены постоянно снижать 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. Поэтому задача повышения качества является одной из самых актуальных в сфере информационных технологий.

1.1 Понятие качества

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” дается следующее определение качества:

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

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: Модель качества программного обеспечения

1.2 Верификация и валидация

Верификация и валидация являются видами деятельности, направленными на контроль качества программного обеспечения и обнаружение ошибок в нем.

Верификация проверяет соответствие одних создаваемых в ходе разработки и сопровождения ПО артефактов другим, ранее созданным или используемым в качестве исходных данных, а также соответствие этих артефактов и процессов их разработки правилам и стандартам.

Валидация проверяет соответствие любых создаваемых или используемых в ходе разработки и сопровождения ПО артефактов нуждам и потребностям пользователей и заказчиков этого ПО, с учетом законов предметной

области и ограничений контекста использования ПО.

На рис 2 приведено различие между верификацией и валидацией.

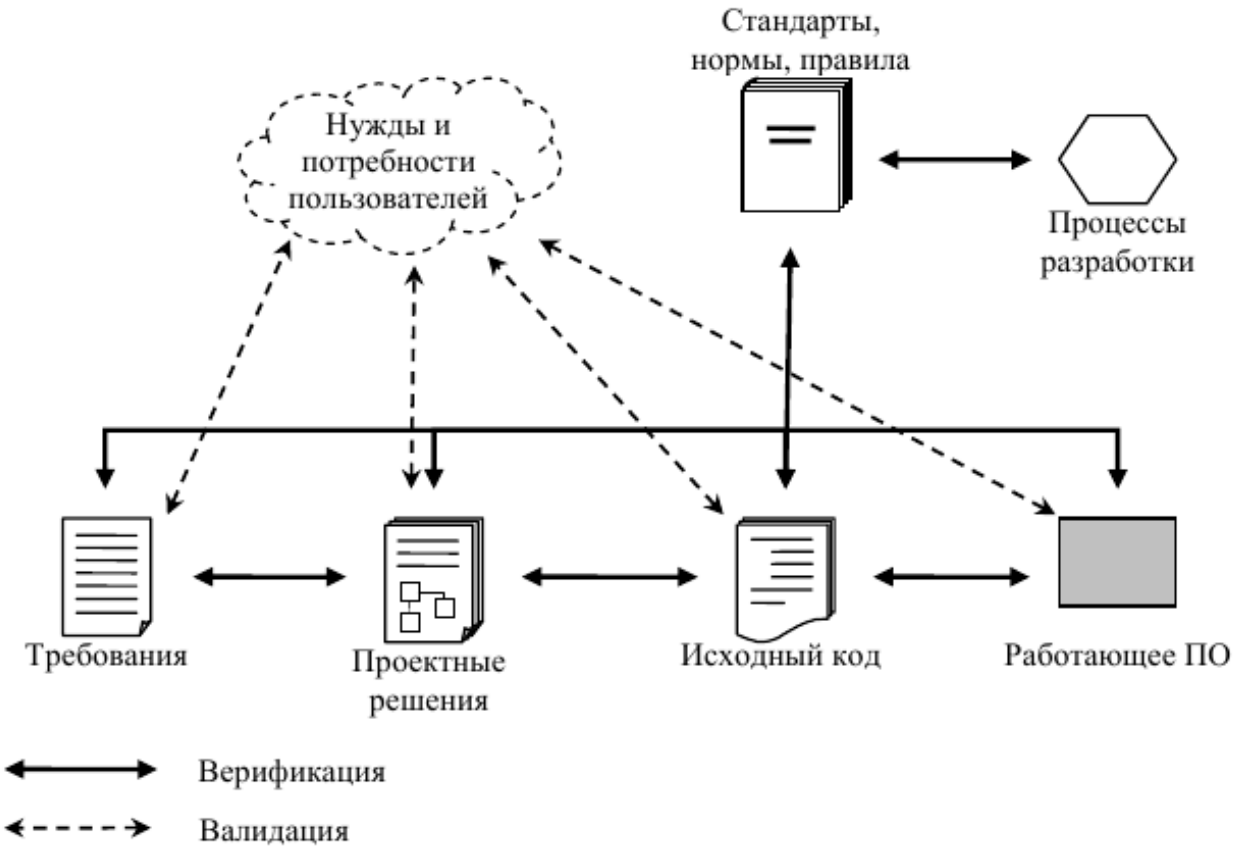

Рис. 2: Соотношение верификации и валидации

1.3 Задачи верификации

Верификация решает следующие задачи в процессе разработки ПО:

- Выявление дефектов различных артефактов разработки ПО.
- Выявление критичных и наиболее подверженных ошибкам частей создаваемой или сопровождаемой системы.
- Контроль и оценка качества ПО.
- Предоставление всем заинтересованным лицам информации о текущем состоянии проекта и характеристиках его результатов.

- Предоставление руководству проекта и разработчикам информации для планирования дальнейших работ, а также для принятия решений о продолжении проекта, его прекращении или передаче результатов заказчику.

2 Методы обеспечения качества программного обеспечения

Существует две основных группы подходов к разработке качественного программного обеспечения.

2.1 Подходы, основанные на синтезе ПО

Данная группа основывается на использовании различных формализаций и модельных представлений во время проектирования архитектуры программной системы. Таким образом, путем дополнительных усилий на начальном этапе разработки продукта, можно минимизировать возможность появления ошибок в дальнейших этапах жизненного цикла.

В данном подходе применяются:

- 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.

2.2 Подходы, основанные на анализе уже созданного ПО

Данная группа подходов предназначена для повышения качества уже созданного ПО. Актуальность этой задачи чрезвычайно высока, так как к данному моменту уже создано огромное количество программных систем, многие из которых имеют проблемы с уровнем качества, которые проявляются в виде различных ошибок и сбоев.

Предполагается, что уже имеется разработанное программное обеспечение, и необходимо оценить и повысить его качество. Проверка может заключаться либо в доказательстве того, что программа соответствует предъявленным функциональным и нефункциональным требованиям, либо в приведении контрпримеров, показывающих несоответствие программы этим требованиям.

2.3 Классификация методов обеспечения качества

Обычно выделяют следующие базовые классификации методов обеспечения качества:

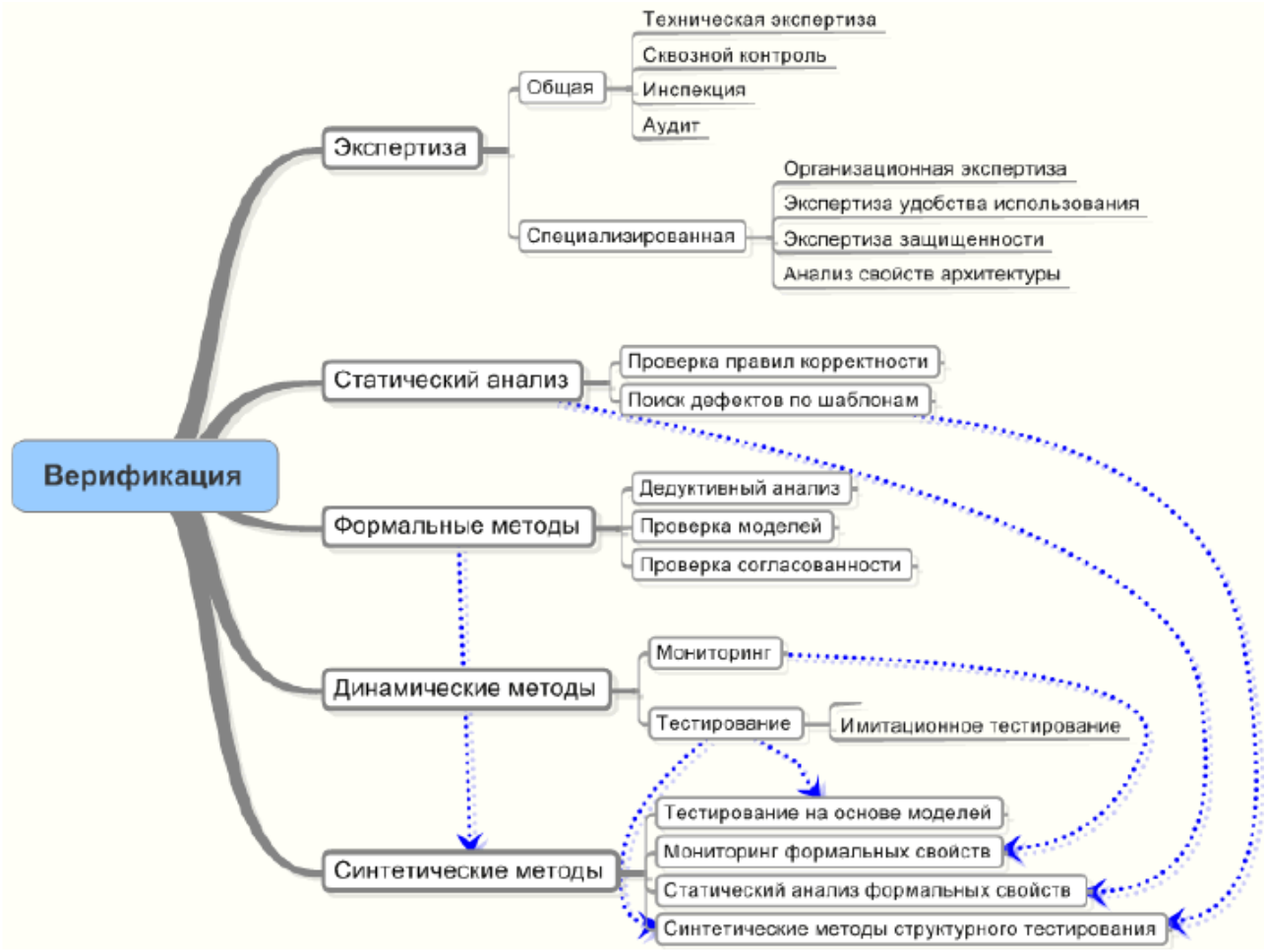

Рис. 3: Схема используемой классификации методов верификации

2.3.1 Экспертиза

От других методов верификации экспертизу отличает возможность выполнять ее, используя только сами артефакты жизненного цикла, а не их модели или результаты работы, как в формальных и динамических методах. Она позволяет выявлять практически любые виды ошибок, причем делать это на этапе подготовки соответствующего артефакта. В то же время она не может быть автоматизирована и требует активного участия людей.

2.3.2 Статический анализ

Статический анализ - набор методов, направленный на статический поиск ошибок в исследуемой программе.

От остальных методов верификации его отделяет то, что статический анализ позволяет обнаруживать ошибки, вносимые на стадии кодирования. Это связано с тем, что среди анализируемых артефактов отсутствует спецификация программы, а это значит, что анализатор ничего не может знать о том, что делает программа. Однако, благодаря этому, статический анализ можно полностью автоматизировать.

2.3.3 Формальные методы

Данные методы использует формальные модели требований, поведения ПО и его окружения для анализа свойств ПО. 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.

2.3.4 Динамические методы

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.

2.3.5 Синтетические методы

Данные 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) и мониторинг формальных свойств (runtime verification). Цель таких методов - объединить преимущества уже используемых подходов.

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.

3.1 Виды моделей программных систем

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

- 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.

3.1.1 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.

3.1.2 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.

3.1.3 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.

4 Использование фреймворков для анализа программ

Так как сложность и количество разрабатываемых программных систем постоянно растет, то для поддержания требуемого уровня качества все чаще начинают использоваться специализированные инструментальные средства (фреймворки). Данные средства поддерживают широкий набор инструментов для проведения анализа:

- Построение метрик.
- Построение моделей программ и применение различных алгоритмов над этими моделями.
- Интерактивная визуализация процесса анализа.

4.1 Структура фреймворков для анализа программ

Обычно фреймворки строятся по модульному принципу и позволяют пользователю комбинировать используемые подходы, а также добавлять свои собственные.

На рис 4 изображена типичная упрощенная структура таких фреймворков:

Рис. 4: Упрощенная структура фреймворка для анализа программ

1. Входными данными могут являться исходный код анализируемой программы или какие-то метаданные, описывающие программную систему.

2. Преобразователь позволяет привести входные данные к виду, удобному для проведения дальнейшего анализа. Эта часть фреймворка является расширяемой - пользователь может подключать различные преобразователи для анализа соответствующих видов входных данных.

3. Метамодель является промежуточным представлением анализируемой системы. Метамодель должна обладать следующими свойствами:

   - Быть независимой от какого-либо языка программирования.
   - Обладать необходимой полнотой представления для проведения различных видов анализа.
   - Содержать обратную связь с исходными входными данными.

   Метамодели могут поддерживать несколько парадигм программирования, однако чаще всего встречаются метамодели, поддерживающие только объектно-ориентированное программирование.

4. После построения метамодели пользователь может использовать ее для проведения интересующих его видов анализа. Причем алгоритмы анализа могут как входить в состав реализации фреймворка, так и подключаться извне.

5. Результаты анализа обычно предоставляются в графической или текстовой форме.

4.2 Анализ существующих решений

4.2.1 SMILE

SMILE - фреймворк для построения метрик программных систем и обладает следующими характеристиками:

1. Независим от языка программирования, на котором написана анализируемая система.
2. Поддерживает большое количество метрик.
3. Поддерживает анализ различных версий анализируемой системы.

В качестве метамодели SMILE использует представление в виде eCST (enriched Concrete Syntax Tree), которая представляет собой дерево разбора программы с добавлением универсальных узлов, что позволяет сделать дерево разбора независимым от входного языка программирования.

На рис 5 изображена архитектура данного фреймворка:

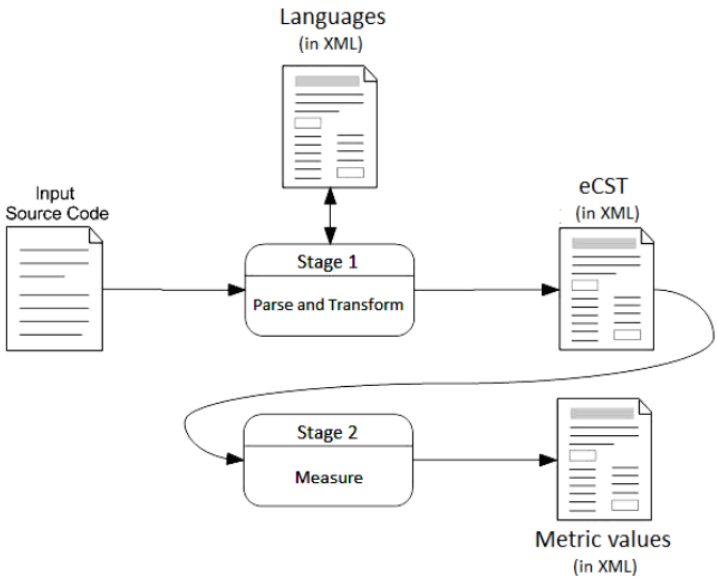

Рис. 5: Архитектура фреймворка SMILE

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.

На данный момент фреймворк находится на стадии разработки и поддерживает малое количество метрик и языков программирования.

4.2.2 Moose

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

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, упрощенная вид которой приведен на рис 6

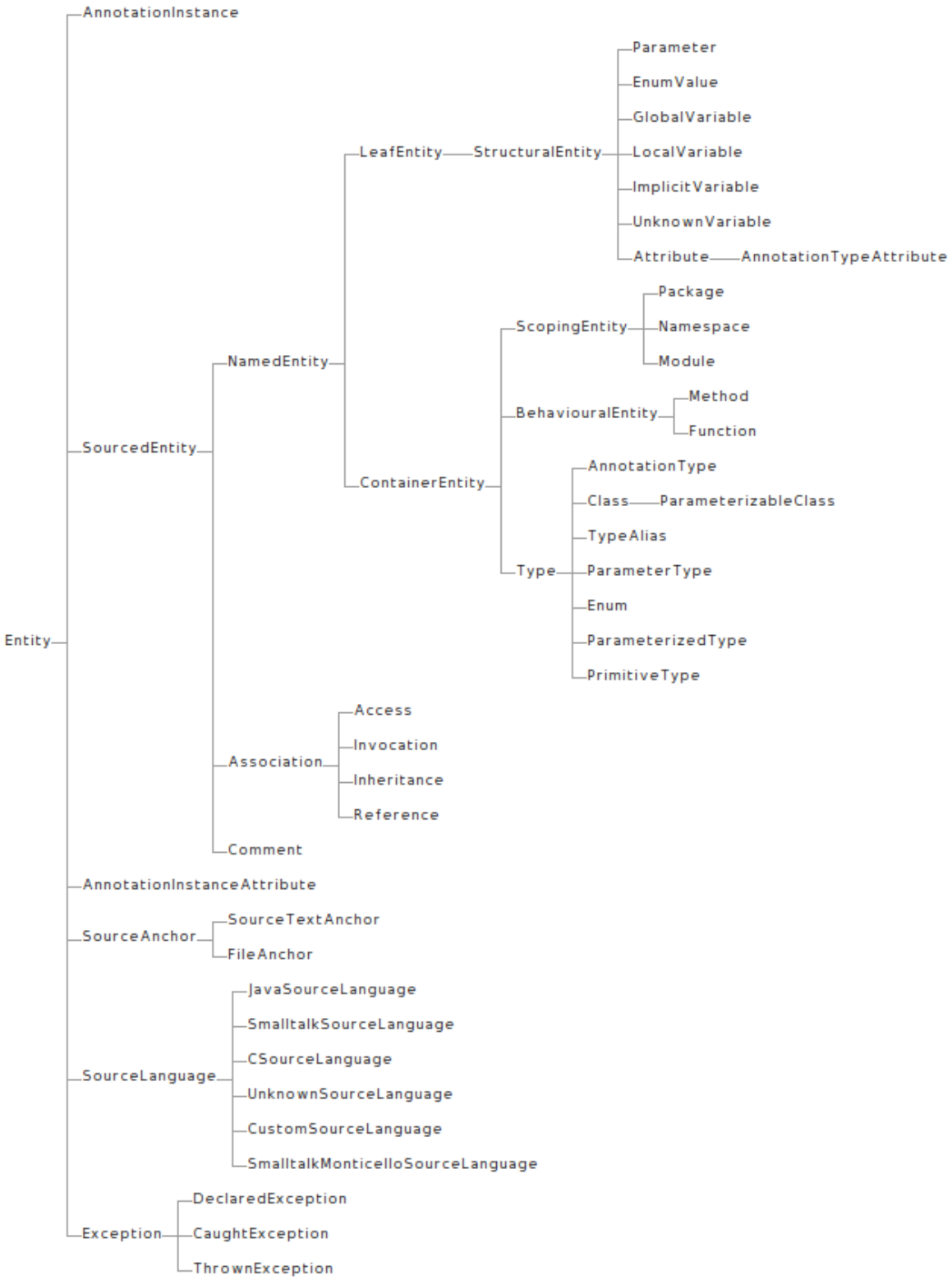

Рис. 6: Структура метамоделей семейства FAMIX

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 фреймворка приведена на рис 7

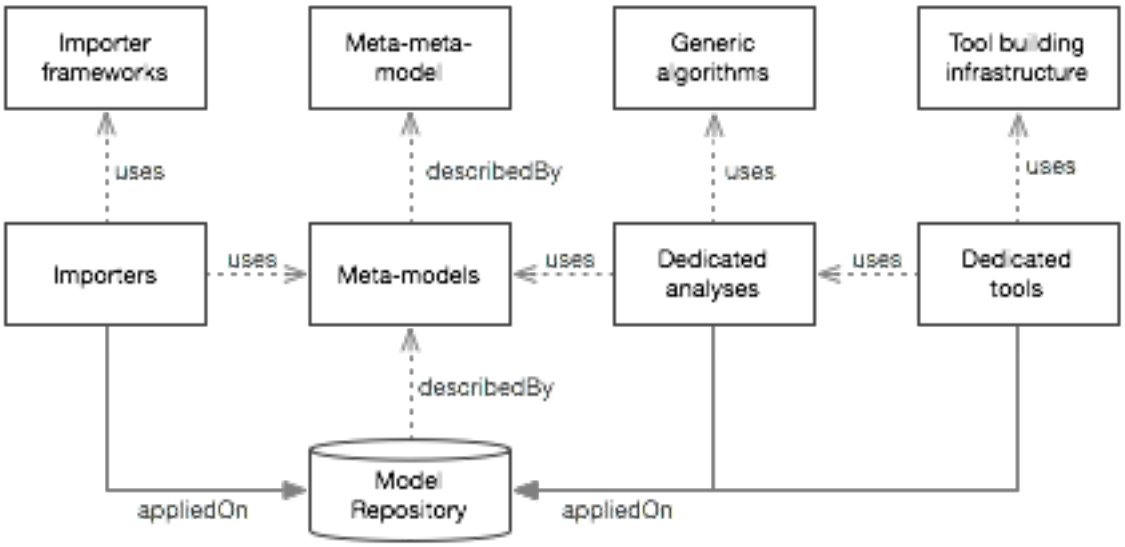

Рис. 7: Архитектура фреймворка Moose

Разработка Moose активно ведется с 1996 года и на данный момент этот фреймворк является одним из самых совершенных средств для анализа программ.

4.2.3 LLVM

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

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.

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 8

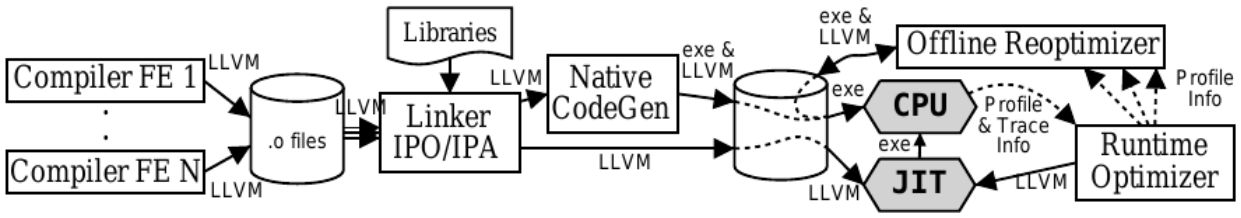

Рис. 8: Архитектура LLVM

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.

5 Постановка задачи

Требуется создать среду для анализа программ, удовлетворяющую следующим требованиям:

1. Предоставлять API на языке Java для проведения анализа, а именно для построения метрик и моделей, перечисленных в главе 3.
2. Иметь модульную структуру - позволять подключать пользовательские алгоритмы анализа.
3. Отображать полученные результаты в графическом виде.

После анализа существующих решений было выяснено, что ни один из рассмотренных фреймворков не удовлетворяет всем перечисленным требованиям, а именно:

- SMILE не обладает достаточной зрелостью, чтобы использовать его в качестве готового решения.
- Moose в основном предназначен для построения метрик программной системы и плохо подходит для решения задач верификации.
- LLVM предоставляет API только для языков C++ и Ocaml.

Исходя из всего вышеперечисленного было принято решение о написании собственного фреймворка, удовлетворяющим указанным требованиям.

6 Заключение

Разрабатываемый фреймворк призван облегчить анализ и инструментирование программных систем путем автоматизации процессов, характерных для многих алгоритмов анализа, например построения графа зависимостей по данным или абстрактного синтаксического дерева.

Предполагается, что в дальнейшем разработанное средство поможет ускорить процесс разработки новых анализаторов, а, благодаря модульной структуре, он будет подходить для решения широкого спектра пользовательских задач.

Список литературы

[1] М.И. Глухих, В.М. Ицыксон. Программная инженерия. Обеспечение качества программных средств методами статического анализа. — Санкт Петербург : Издательство Политехнического университета, 2011.

[2] В.В. Кулямин. Методы верификации программного обеспечения. — Институт системного программирования РАН, 2008.

[3] И. Тетерук, А. Булат. Качество программного обеспечения. — `http://www.protesting.ru/qa/quality.html`.

[4] 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.

[5] 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.

[6] Girba Tudor. The Moose book. — `http://www.themoosebook.org/book`. — 2011.