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, 27 стр., 6 рис.

ABSTRACT

Report, 27 pages, 6 figures

СОДЕРЖАНИЕ

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, 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 разделе 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 разделе 1.1.

Обычно эти задачи решаются вручную каждый раз при создании анализаторов или проведения верификации программы. В данной работе предлагается способ автоматизации решения данных задач на основе метамоделирования.

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]:

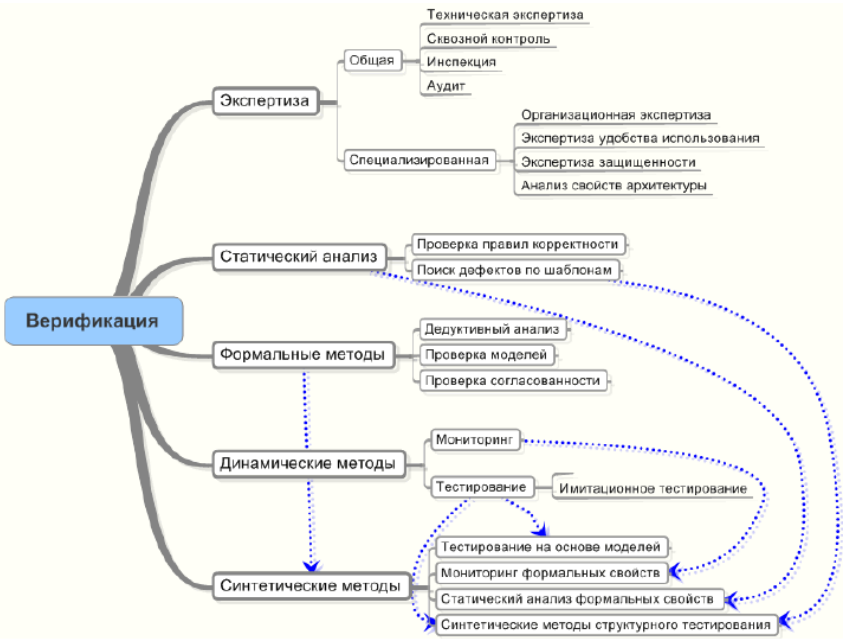

Рисунок 1.1. Схема используемой классификации методов верификации

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.

1.4 Постановка требований к инструментальной среде

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

Таким образом, 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.

Исходя из вышеперечисленного, разрабатываемое средство должно отвечать следующим сценариям использования:

1. Библиотека для извлечения моделей, необходимых для написания процедур статического анализа и верификации
2. Инструментальная среда для визуализации полученных моделей

1.5 Анализ существующих решений

Решение проблем, перечисленных в п. 1.4 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 для обобщения различных алгоритмов анализа. 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

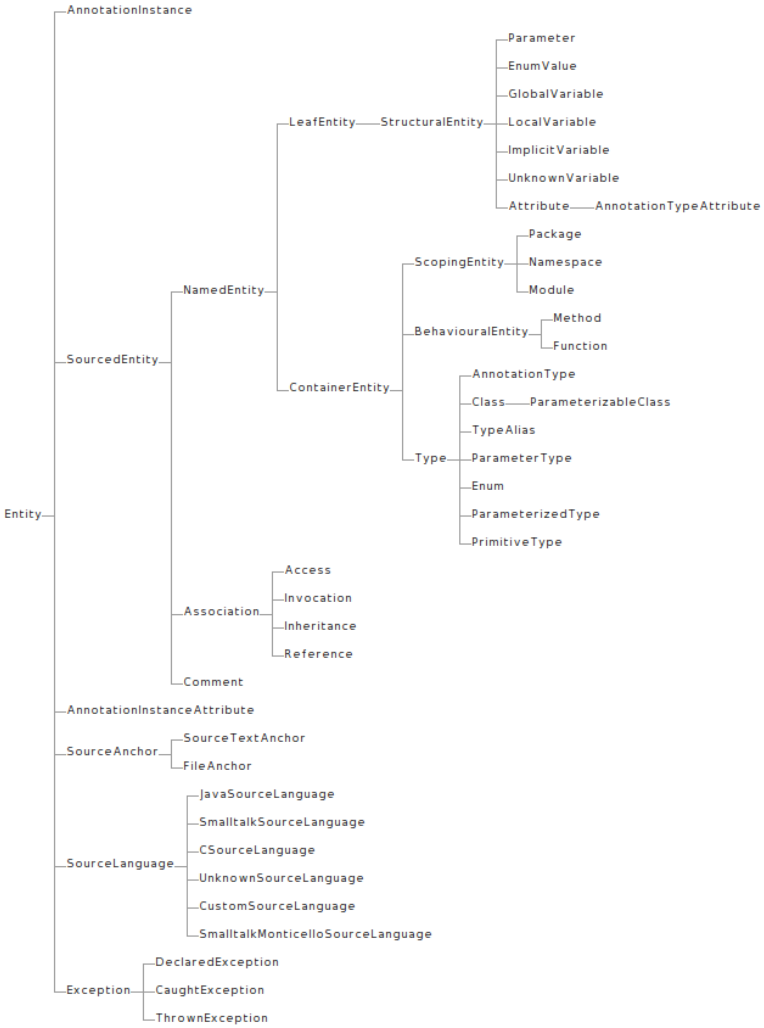

Рисунок 1.2. Структура метамоделей семейства FAMIX

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

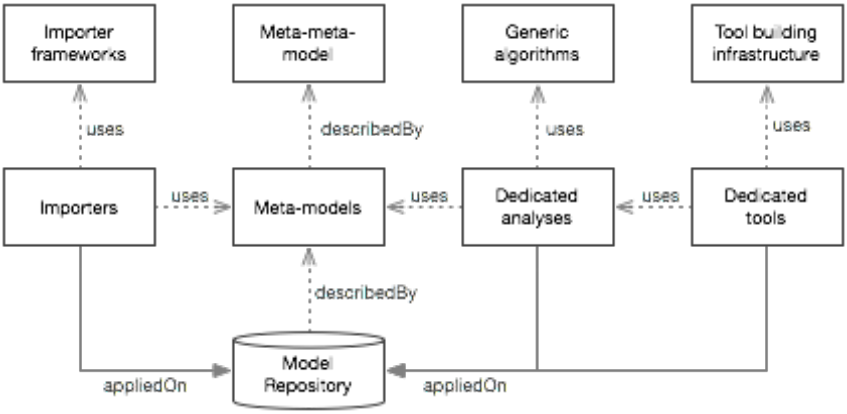

Рисунок 1.3. Архитектура фреймворка Moose

К недостаткам данной среды по отношению к требованиям, поставленным в п. 1.4, 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.

Smile

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

В качестве метамодели SMILE 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:

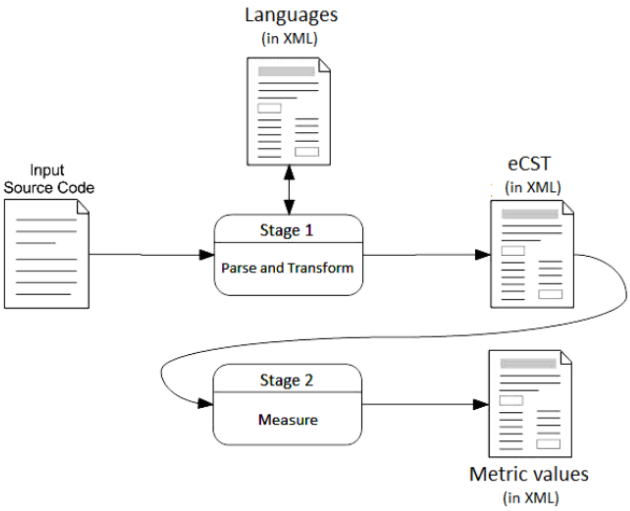

Рисунок 1.4. Архитектура SMILE

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.

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:

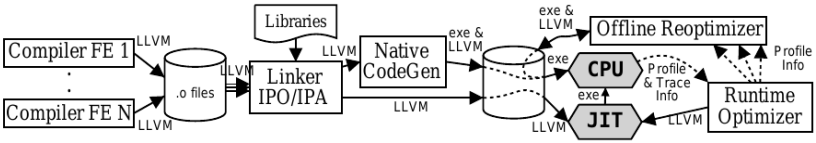

Рисунок 1.5. Архитектура LLVM

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:

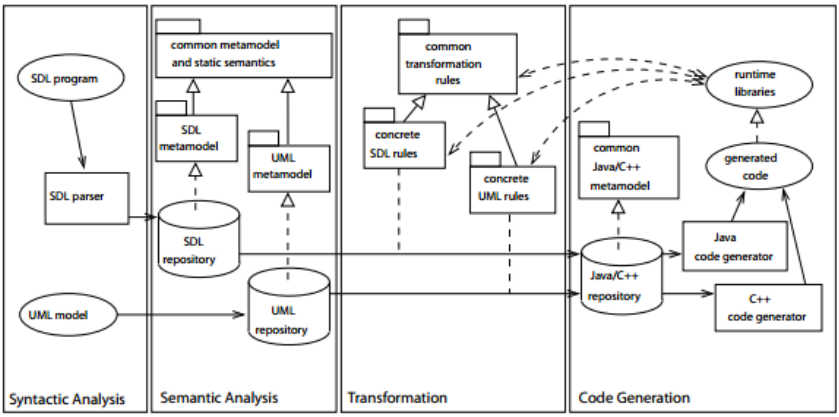

Рисунок 1.6. Архитектура ULF-Ware

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++.

Таким образом, были рассмотрены существующие варианты средств, использующих подход, основанный на языконезависимых метамоделях. Как видно из обзора, ни одно из этих средств не удовлетворяет полностью требованиям, приведенным в п. 1.4.

2 ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ИНСТРУМЕНТАЛЬНОЙ СРЕДЫ

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.