УДК 004.41

М.В. Зубов, А.Н. Пустыгин, Е.В. Старцев

Применение универсальных промежуточных представлений для статического анализа исходного программного кода

Описывается использование промежуточных представлений исходного кода для выполнения статического анализа. Предлагается расширить использование универсальных и многоуровневых представлений в одном общем. Представлен прототип программного инструмента, генерирующий и обрабатывающий такое представление.

Ключевые слова: статический анализ, промежуточное представление, машинные комментарии, исходный код, диаграмма классов, Java, Python.

Статический анализ облегчает выполнение типовых задач разработки. Эти задачи могут быть связаны с поиском ошибок, улучшением качества кода, внутренним тестированием [1]. Перспективным направлением является извлечение информации по исходному коду (построение машинных комментариев [2]).

Промежуточные представления исходных текстов программ. Промежуточное представление (далее – представление) программного текста – это синтаксически и семантически эквивалентный исходному коду набор данных, над которым выполняется анализ [3]. Представление может быть выполнено в виде различных наборов данных. Самый простой и наиболее изученный в области компиляции вариант – абстрактное синтаксическое дерево AST, фактически это результат синтаксического разбора. Более сложной, обеспечивающей увеличение скорости анализа является реляционная база данных. Однако затраты на получение такого представления растут. Один из наиболее эффективных видов представления – конечный автомат. Помимо AST, в компиляции широкое применение нашло представление SSA (Single-State Assignment) и его развитие GSA [4].

Представления, описывающие только один входной язык, будем называть частными. Те, что пригодны для исходного кода на нескольких входных языках, – универсальными. К частным представлениям можно отнести уже упомянутое абстрактное синтаксическое дерево, описывающее исходный текст программы во всех деталях, свойственных конкретному языку программирования. К универсальным представлениям можно отнести SSA и GSA с той оговоркой, что версии переменных будут описываться в терминах конкретного языка.

Использование универсальных представлений сокращает общие затраты при анализе, так как общее количество требуемых функциональных модулей представления и реализации цели анализа для представления снижается. При использовании частных представлений потребуется общее количество ($N + N \times M$) операций, а при использовании универсального представления – $N + M$. При положительных значениях N и M, что имеет место для количества целей и входных языков, первая величина всегда больше второй на $M \times (N - 1)$ операций, что определяет выигрыш в сокращении общего количества операций для покрытия всех входных языков и целей анализа.

Безусловно, трудозатраты зависят не только от общего количества операций но и от их сложности, которая при использовании универсального представления может быть выше (особенно в области генерации универсального представления), но при увеличении количества языков и целей анализа преимущество в $M \times (N - 1)$ операций играет все большую роль. Уже для 3 языков и 3 целей анализа, когда выигрыш достигает уменьшения операций в 2 раза, имеется серьезная причина для выбора в пользу универсального представления. Из основного достоинства универсального представления следует и недостаток: универсальные представления не позволяют отразить все детали синтаксиса каждого из языков исходного текста программ.

Применение универсальных и многоуровневых представлений. Для выполнения анализа нам вовсе не обязательно иметь полную информацию обо всех деталях исходного текста. Самым простым уровнем является наиболее детализированный. Он полностью соответствует исходному коду. В различных проектах выделяют разное количество уровней. Так, например, в анализаторе PQL предлагается применять 3 уровня: модель глобального программного дизайна, модель структуры программы, модель детального программного дизайна [5]. А в Bauhaus Project имеется 2 уровня:

низкоуровневый Inter Mediate Language, детализированный до операторов, и Resource Flow Graphs, описывающий архитектуру [6]. Нечто подобное наблюдается и в организации компилятора GCC, в котором используются представления GIMPLE, CFG (Control Flow Graph), RTL, помимо классического AST [7]. Как видно, различные подходы сходятся в том, что есть полностью соответствующее коду представление и описывающее глобальную структуру программы, архитектуру. Между этими двумя крайними уровнями можно выделить дополнительные уровни представлений. С точки зрения анализа основной интерес представляют структурные единицы используемой парадигмы программирования и языка. Например, для объектно-ориентированных языков это может быть представление уровня классов, их иерархии и взаимосвязей (похоже на диаграмму классов UML [8]).

XML-форма классового представления. Для прототипной реализации были выбраны весьма разные языки: Java и Python. Самое главное их отличие в том, что Java – это язык со строгой типизацией данных, а Python – с динамической. Это накладывает некоторые ограничения на получение связей агрегирования и композиции, потому что типы полей в Python определяются на этапе исполнения, а не на этапе компиляции, т.е., они не могут быть получены напрямую из исходного кода. Для упрощения получения прототипа откажемся от получения типов и построения связей агрегирования.

В Python присутствует множественное наследование, которого нет в Java. В Java есть интерфейсы, которых нет в Python, но в Java есть множественное наследование у интерфейсов, а также класс может реализовывать несколько интерфейсов. В данном случае самым простым решением будет приравнять интерфейсы к классам. Фактически, так и есть, интерфейс – это абстрактный класс, который не содержит никаких реализаций. Таким образом, представление будет допускать множественное наследование для всех входных языков, даже для тех, где его нет. Отказываясь от интерфейсов, также приходится отказываться от отношений реализации. В Python'e такое отношение не может быть реализовано средствами языка так, чтобы оно отличалось от отношения обобщения. Однако оно может присутствовать на уровне логики. Для предлагаемого представления удобно использовать древовидные структуры. В качестве формата данных хорошо подойдет XML. В качестве основного тега выступает тег Class, описывающий отдельный класс, в качестве дочерних тегов используются теги: Attr – описывает поля класса; Method – описывает методы класса; Parent – описывает родителей класса.

Графическое представление XML-схемы [9] изображено на рис. 1. Полный текст схемы можно получить из git-репозитория [10].

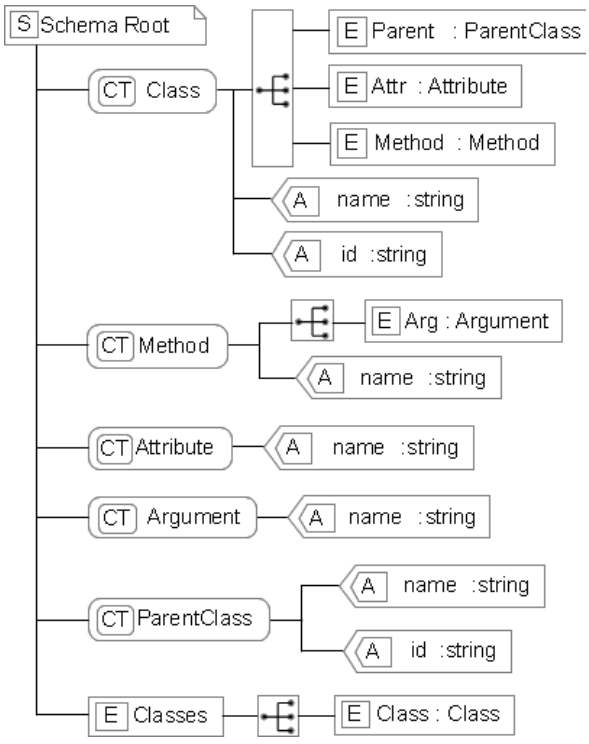

Рис. 1. Графический вид XML-схемы универсального классового представления

Прототип системы получения универсального классового представления. Реализованный прототип поддерживает 2 входных языка программирования – Java и Python. Для Python использовался генератор на основе библиотеки logilab-astng, что позволило сразу получать информацию о классах проекта. Для Java использовался генератор, использующий AST. Дерево разбора получалось из XML-документа, генерируе-мого доработанным компилятором Open JDK. Реализация прототипа выложена в git-репозитории на открытый ресурс GitHub под свободной лицензией.

Результаты получения представления и его визуальной формы. Для тестирования предложенного промежуточного представления была получена визуализированная иерархия классов в dot-формате описания графов из пакета Graphviz, так как иерархия наследования может быть представлена в виде направленного графа.

Для демонстрации универсальности представления и его полноты, независимо от входного языка, возьмем типовую реализацию шаблона проектирования «Посетитель» [11]. Были написаны 2 одинаковые по смыслу реализации на Java и Python. Полные тексты примеров доступны в git-репозитории [12].

Листинг 1. Фрагмент промежуточного представления примера на Python.

```
<?xml version='1.0' encoding='utf-8'?>
<Classes>
  <Class id="4" name="Node">
    <Attr name="strPos"/>
    <Attr name="lineno"/>
    <Method name="accept">
      <Arg name="visitor"/>
    </Method>
  </Class>
  <Class id="5" name="VariableNode">
    <Attr name="name"/>
    <Method name="accept">
      <Arg name="visitor"/>
    </Method>
    <Parent id="4" name="Node"/>
  </Class>
  <Class id="11" name="NodeVisitor">
    <Method name="visit_variable">
      <Arg name="node"/>
    </Method>
    <Method name="visit_assignment">
      <Arg name="node"/>
    </Method>
    <Method name="visit_class">
      <Arg name="node"/>
    </Method>
  </Class>
  <Class id="12" name="Verificator">
    <Attr name="specification"/>
    <Method name="visit_assignment">
      <Arg name="node"/>
    </Method>
    <Method name="visit_variable">
      <Arg name="node"/>
    </Method>
    <Method name="visit_class">
      <Arg name="node"/>
    </Method>
    <Parent id="11" name="NodeVisitor"/>
  </Class>
```

Промежуточное представление иерархии классов для Python в виде XML представлено в листинге 1. Для Java визуализация такого представления в UML-подобной форме представлена на рис. 2. Анализируя данные, видно, что диаграмма и XML-представление идентичны по смыслу.

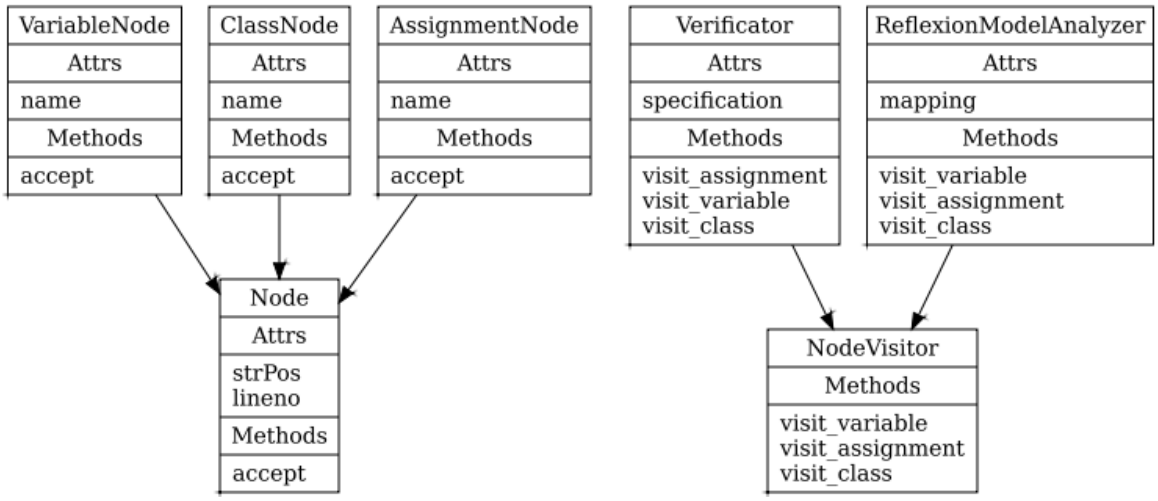

Рис. 2. UML-подобная диаграмма классов примера Java

Анализ иерархии классов с помощью построенного прототипа. Объектная структура программного продукта в соответствии с современными представлениями должна удовлетворять определённым критериям, в том числе правильно построенным линиям наследования от базовых классов.

С целью тестирования возможностей построенного прототипа было проверено выделение методов для групп объектов, которые могут быть вынесены в общий суперкласс.

Для анализа были выбраны несколько Python- и Java-проектов. На Python – библиотека для веб-разработки Django версии 1.4.1, сетевая библиотека Twisted версии 12.0.0, Python-пакет logilab, который использовался для обработки AST-деревьев (компоненты logilab-astng версии 0.23.1 и logilab-common версии 0.58.0) (табл. 1). Проблемными методами названы те, чьи имена совпадают в разных классах, но не унаследованы ими от общего суперкласса.

Таблица 1

Результаты анализа Python-проектов

Проект	Всего методов	Проблемных методов
Django	2079	334
Twisted	8651	1291
Logilab	491	110

На Java анализировались компилятор javac, из состава OpenJDK v6u20, статический анализатор FindBugs v2.0.1, SQL-клиент SquirreL (табл. 2).

Таблица 2

Результаты анализа Java-проектов

Проект	Всего методов	Проблемных методов
Javac	2076	397
FindBugs	5538	787
SQuirreL	7798	1324

Кроме того, был проанализирован собственный код. Для анализа был выбран построитель AST для Java. В результате анализа было найдено 20 «похожих» методов без родительской реализации из 102. Как оказалось, класс, выполняющий обход узлов, получаемых от парсера, делегировал вызовы зарегистрированным в нем классам, выполняющим вывод в XML или другую обработку. Причем делегировал он их, повторяя интерфейс этих обработчиков в 15 из 17 методов. При такой реализации стоило в самом классе, выполняющем обход, реализовать интерфейс обработчика, так как его поведение совпадало с поведением обработчика.

Таким образом, был проведен рефакторинг собственного кода, сокративший число проблемных методов с 20 до 3, причем, с точки зрения чистоты кода, этот рефакторинг требовался. Из оставшихся трех два также подлежат будущему рассмотрению. С рефакторингом можно ознакомиться по ссылке https://github.com/exbluesbreaker/csu-code-analysis/commit/64c6974517fdc700c2dc9f0c0173208ab3352af5 (информация о коммите в git).

Полученные результаты дают хорошую почву для возможного рефакторинга исходного кода проанализированных программных продуктов, это может быть как выделение новых суперклассов или интерфейсов, так и добавление новых методов к уже существующим. Доля имен методов с одинаковыми именами, реализованными в разных классах, более 10% для всех исследованных продуктов, а для некоторых она достигает почти 20%. Разработанный прототип не предназначен для автоматизации такого рефакторинга, он лишь извлекает из исходного кода полезную для разработчика информацию.

Выводы. В результате проделанного исследования была разработана и описана модель промежуточного представления, позволяющая универсальным инструментом выполнять статический анализ исходного кода Java и Python. Представлен прототип такого инструмента. Для другого языка следует только получить универсальное промежуточное представление по предлагаемому шаблону.

Литература

1. Зубов М.В. Статический анализ ПО с помощью его промежуточных представлений и технологий с открытым исходным кодом / М.В. Зубов, А.Н. Пустыгин, Е.В. Старцев // Матер. 2-й Междунар. конф. «FOSS. Lviv–2012», Львов. – Львів: Сорока, 2012. – С. 165–168.

2. Автоматический синтез комментариев к программным кодам: перспективы развития и применения / А.Н. Пустыгин, А.И. Иванов, Ю.К. Язов, С.В. Соловьев // Программная инженерия (Москва). – 2012. – № 3. – С. 30–34.

3. Зубов М.В. Подходы к статическому анализу открытого исходного кода / М.В. Зубов, А.Н. Пустыгин, Е.В. Старцев // Матер. 8-й Междунар. конф. «Linux Vacation» Eastern Europe. – Брест: Альтернатива, 2012. – С. 36–40.

4. The Formation and Simulation of a «Whole Program» Gated Singular Assignment Program Dependence Graph / J. Cook, D. Gottlieb, B. Greskamp, R. Kujoth. – Unconventional Computer Architecture

Group, Final Report. – 2008 [Электронный ресурс] – Режим доступа: http://iacoma.cs.uiuc.edu/~greskamp/pdfs/497mf.pdf, свободный (дата обращения: 15.01.2013).

5. Jarzabek S. Design of Flexible Static Program Analyzer with PQL // IEEE Transactions on software engineering. – 1998. – Vol. 24, № 3. – С. 197–215.

6. Bauhaus project [Электронный ресурс]. – Режим доступа: mtc.epfl.ch/software-tools/blast/index-epfl.php, свободный (дата обращения: 02.12.2012).

7. Basic gcc Intermediate Representation Dumps [Электронный ресурс]. – Режим доступа: www.cse.iitb.ac.in/~uday/courses/cs324-05/gccProjects/node4.html, свободный (дата обращения: 02.12.2012).

8. Буч Г. Язык UML. Руководство пользователя / Г. Буч, Дж. Рамбо, А. Якобсон. – СПб.: Питер, 2004. – 432 с.

9. W3C XML Schema Definition Language (XSD) 1.1 [Электронный ресурс]. – Режим доступа: www.w3.org/TR/xmlschema11-1, свободный (дата обращения: 02.12.2012).

10. XML-схема классового представления [Электронный ресурс]. – Режим доступа: https://github.com/exbluesbreaker/csu-code-analysis/blob/master/data/classes.xsd, свободный (дата обращения: 02.12.2012).

11. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб.: Питер, 2007. – 366 с.

12. GitHub. Репозиторий CSU-code-analysis [Электронный ресурс]. – Режим доступа: https://github.com/exbluesbreaker/csu-code-analysis, свободный (дата обращения: 02.12.2012).

Зубов Максим Валерьевич
Аспирант каф. компьютерной безопасности и прикладной алгебры (КБиПА) ЧелГУ, г. Челябинск
Тел.: +7-961-784-45-31
Эл. почта: zubovmv@gmail.com

Пустыгин Алексей Николаевич
Канд. техн. наук, д доцент каф. КБиПА
Тел.: +7-905-835-98-68
Эл. почта: p2008an@rambler.ru

Старцев Евгений Владимирович
Аспирант каф. КБиПА
Тел.: +7-961-789-69-23
Эл. почта: slayer-gurgen@yandex.ru

Zubov M.V., Pustygin A.N., Startsev E.V.

Use of the intermediate software representations for static analysis of source code

This article describes using of intermediate representations of source code for static analysis. It is proposed to extend using of generic and multi-level representations in one common. Software tool prototype that can make and process this kind of representation is introduced.

Keywords: static analysis, intermediate representation, machine comments, source code, class diagram, Java, Python.