*Модел. и анализ информ. систем.* Т. **17**, № **4** (2010) 70–77

УДК [004.4'233 + 004.4'422] : 004.438Java

# Интеграция семантических верификаторов в компиляторы языка Java

Клепинин А. В., Мелентьев А. А.

*Уральский Государственный Университет им.А.М.Горького*

*e-mail: alklepin@mail.utnet.ru amelentev@gmail.com*

*получена 14 октября 2010*

**Ключевые слова:** верификация программ, абстрактное синтаксическое дерево, Java, компилятор, JSR269, javac, ecj, Eclipse IDE, Netbeans IDE.

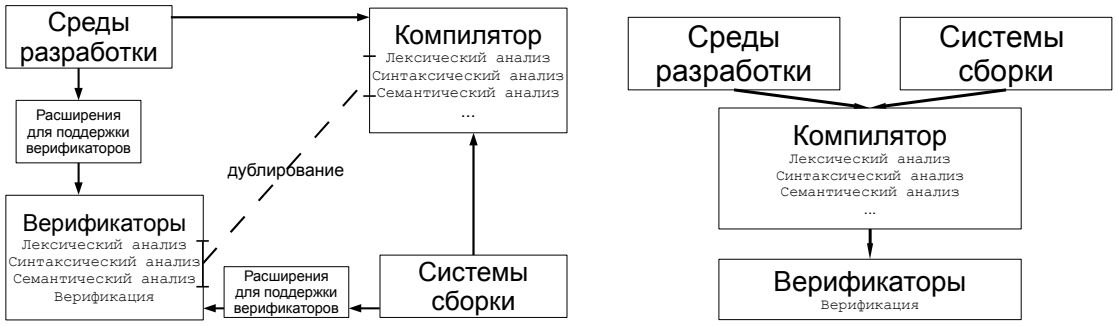
Рассматривается способ статического семантического анализа исходных кодов программы на стадии ее компиляции с целью повышения качества исходного кода. В качестве способа реализации такого семантического анализа предлагается унифицированная интеграция в компиляторы языка Java для получения полного доступа к синтаксическому дереву (AST) компилируемых программ после этапа семантического анализа. Для обеспечения унификации реализованы общие интерфейсы для работы с синтаксическим деревом и адаптеры к реализациям синтаксических деревьев в компиляторах Sun/Oracle javac и Eclipse Compiler for Java (ecj). Выбранный способ обеспечил прозрачную интеграцию со средами разработки Eclipse и Netbeans без необходимости установки каких-либо расширений данных сред. Разработанный метод демонстрируется на некоторых примерах верификации программ.

## 1. Введение

В процессе разработки ПО обычно используются универсальные языки программирования. Но в конкретных задачах часто уместно сузить возможности применимости отдельных конструкций языка, поскольку иначе они могут приводить к гарантированным ошибкам. Хотелось бы иметь возможность выполнять дополнительный семантический контроль программ и выявлять потенциально опасные или запрещённые в данном проекте конструкции.

Для общих проверок часто применяются различные внешние средства для проверки корректности программ (внешние статические верификаторы; например, для языка Java это — pmd [1], findbugs [2, 3]). Но для проверки специфичных областей все равно приходится писать собственные верификаторы. Также можно заметить (см. рис. 1a), что первые фазы работы внешних верификаторов, основанных на анализе исходного кода (лексический, синтаксический и поверхностный семантический

анализы), такие же, как и в компиляторе. Зачем запускать их заново, если можно воспользоваться результатами работы компилятора? Кроме того, для удобства использования внешних верификаторов необходимо обеспечивать их интеграцию с различными средами разработки и системами сборки. А если встроить верификаторы в компилятор, то можно об этом не беспокоиться. И среды разработки, и системы сборки будут автоматически запускать верификацию через компилятор без необходимости установки расширений (см. рис. 1b).




(a) внешние (b) встроенные в компилятор

Рис. 1: Верификаторы

Здесь также следует отметить и человеческий фактор. Встроенные средства диагностики ошибок требуют от программиста немедленной реакции, в то время как внешние средства, встроенные в процесс сборки продукта, выявляют проблемы на более поздней стадии, что приводит к увеличению времени на устранение проблемы. Разработчикам нужно видеть ошибки прямо в процессе разработки, а не дожидаться длительного процесса сборки или вручную запускать внешние верификаторы.

Выявлять ошибки можно и во время исполнения программ [4] (динамическая верификация), но процесс этот гораздо более сложный и затратный, нежели выявление ошибок при компиляции. Во-первых, для этого требуется специальная отладочная сборка со встроенными проверками во время выполнения (соответственно есть вероятность, что ошибка проявится в финальной (release) версии, но не в отладочной). И во-вторых, требуется отдельный тестовый запуск программы для обнаружения ошибок (при этом нет гарантии, что он обнаружит все ошибки; некоторые пути исполнения программы могут быть не учтены в отличие от статического анализа, который проверяет все возможные варианты). Стоит заметить, что множества ошибок, выявляемых статической и динамической верификацией, несравнимы как множества. То есть при разработке нежелательно ограничиваться лишь одним из видов верификации. И разработка удобного процесса статической верификации, совмещённого с процессом компиляции, представляется разумной и актуальной целью.

## 2. Постановка задачи

Одним из способов обеспечить семантический анализ на стадии компиляции является интеграция с компилятором с целью обеспечения верификатору доступа к синтаксическому дереву, формируемому в процессе компиляции программы. Обычно существуют несколько версий компиляторов для одного языка. Например, для языка Java наиболее известные компиляторы – это классический javac, входящий в состав Java Developer Kit и используемый в среде разработки Netbeans IDE, и Eclipse Compiler for Java (ECJ), используемый в среде разработки Eclipse IDE. В разных компиляторах зачастую применяются совершенно разные алгоритмы и, соответственно, деревья разбора. Хотелось бы, чтобы верификаторы работали с любым компилятором. В случае Eclipse и Netbeans IDE это бы автоматически означало, что сообщения о проблемах в коде появлялись бы прямо в среде разработки с указаниями места возникновения и прочей информацией в процессе компиляции без запуска каких-либо внешних утилит.

Стандарт JSR269: Pluggable Annotation Processing [5] на компиляторы Java определяет интерфейсы, которые предоставляют некоторый доступ для чтения информации из синтаксического дерева компилируемых программ. Но эти интерфейсы отражают только базовую структуру программы и не предоставляют информации о её реализации. Поэтому для нужд верификации JSR269 в чистом виде бесполезен.

Интеграция в компиляторы Java есть в библиотеке Lombok [6], но там пока не решена проблема совместимости компиляторов. Для интеграции специальных обработчиков (расширителей, верификаторов) Lombok с каждым конкретным компилятором пишется свой отдельный код, использующий отдельное семейство классов, привязанных к компилятору. Кроме того, проект ориентирован на расширение возможностей языка (путем модифицирования AST) за счёт применения аннотаций. Lombok использует JSR269 для получения управления при компиляции. В будущих версиях Lombok планируется унификация синтаксических деревьев компиляторов, чтобы можно было работать с одним унифицированным семейством классов, не привязанным к компиляторам.

Подобная интеграция есть и в инструменте статической верификации Checker Framework [7], но там интеграция реализована только для компилятора javac версии 7. Кроме того, Checker Framework ограничен только проверкой типов с использованием аннотаций (@NotNull, @Immutable, @ReadOnly и прочие). Checker Framework также планирует использовать унифицированное AST в будущих версиях [8]. Многие другие верификаторы могут быть адаптированы к унифицированному AST и таким образом существенно расширят область своей применимости.

Целью данной работы является разработка способа верификации программ, совмещённого с процессом компиляции на различных компиляторах и средах разработки для языка Java. Разработка унифицированного дерева разбора, которая необходима для реализации данной цели, сама по себе важна, актуальна и полезна для многих инструментов верификации.

# 3. Результаты

В результате проделанной работы была создана библиотека juast [9] с открытым исходным кодом. С использованием этой библиотеки можно как наложить семантические ограничения на обрабатываемый компилятором код, так и модифицировать синтаксическое дерево программы, реализуя, таким образом, возможность расширения языка программирования. Полученные верификаторы и расширения будут работать везде, где используются компиляторы javac и ecj, будь то среды разработки Eclipse IDE, Netbeans IDE, системы сборки Ant, Maven и прочие инструменты.

Juast подключается к компиляторам и к средам разработки как обычная библиотека (например: `javac -cp juast.jar Test.java`) без какой-либо модификации компилятора и предоставляет унифицированный доступ к деревьям разбора компилируемых программ после этапа семантического анализа (см. рис. 2).

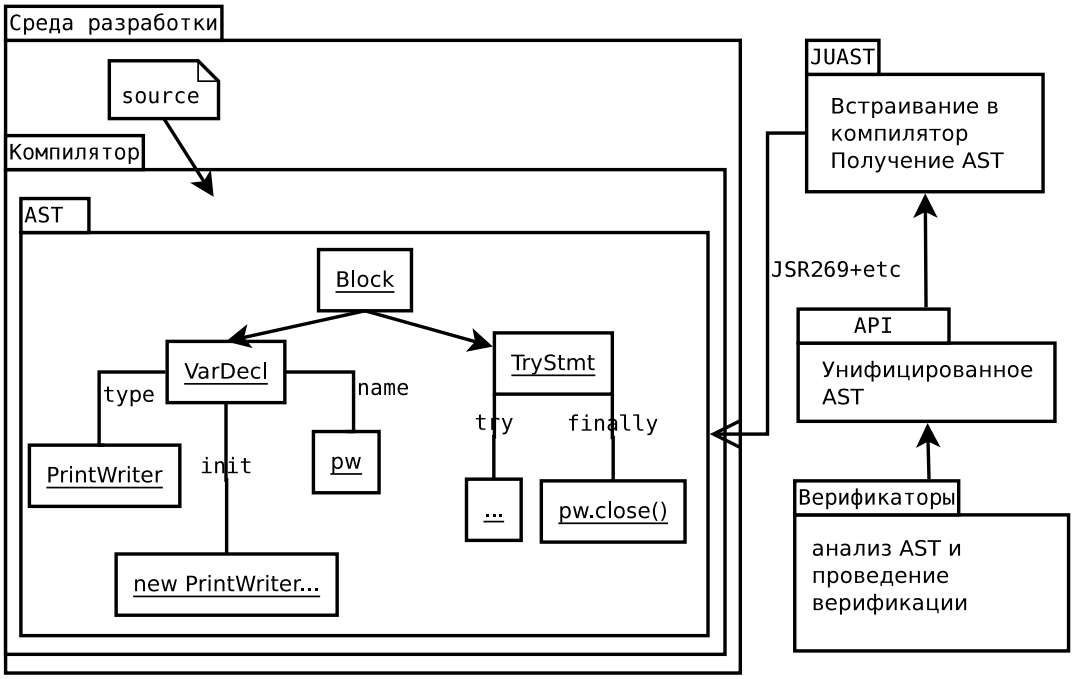



Рис. 2: Схема доступа к компилятору

Интеграция в компиляторы и среды разработки достигается посредством интерфейсов JSR269 [5] и через некоторые недокументированные методы, так как прямого и публичного доступа к деревьям разбора в компиляторах не существует.

Для обеспечения унификации работы с компиляторами javac и ecj (и соответственно средами Netbeans и Eclipse) в библиотеке реализованы интерфейсы унифицированного синтаксического дерева (AST) языка Java и адаптеры к компиляторам. Интерфейсы AST предоставляют доступ к информации о предках (заметим, что в деревьях javac и ecj она отсутствует) и реализуют стандартный шаблон “посетитель” (visitor) для обхода синтаксического дерева.

В процессе обхода синтаксического дерева верификаторы могут проверить кор-

ректность программ и пометить обнаруженные ошибки в узлах дерева. Компиляторы и среды разработки выводят такие сообщения об ошибках в стандартном для них виде с указанием места возникновения. Далее, если критических ошибок не было обнаружено, продолжается стандартный процесс компиляции (оптимизация и генерация кода).

На базе библиотеки juast реализованы некоторые простые верификаторы, иллюстрирующие возможности технологии. Например, реализована проверка того, что внешние ресурсы, используемые программой, корректно закрываются в секции finally соответствующего блока try (см. рис. 3). Также для проверки возможностей расширения компиляторов реализована перегрузка операторов “+”, “-”, “*”, “/” для типов BigInteger и BigDecimal. Подход, в принципе, обобщается до перегрузки операторов в общем виде.

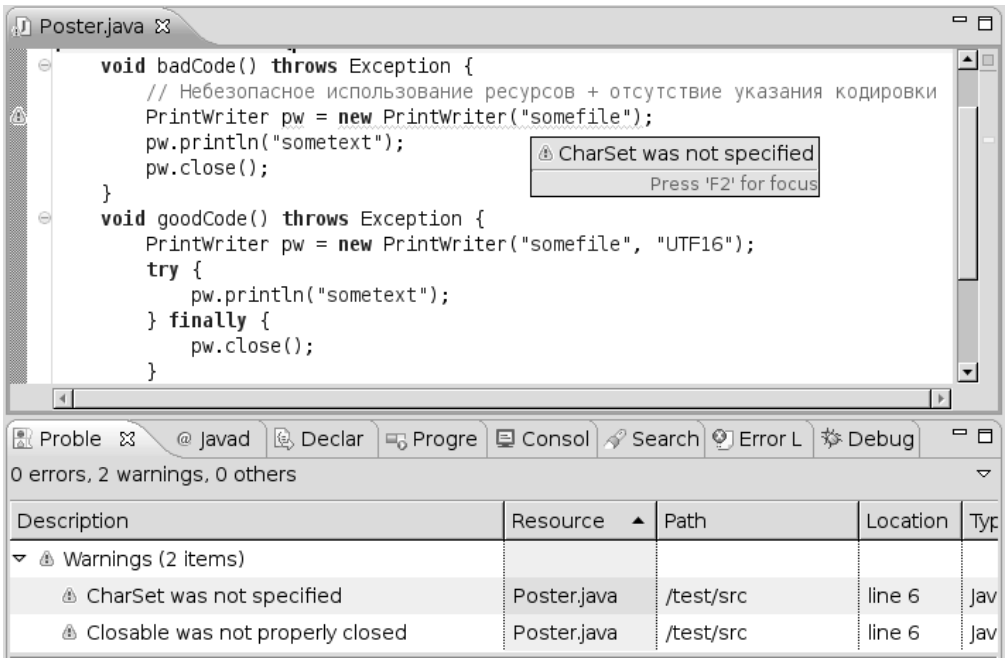



Рис. 3: Эффект библиотеки juast.jar на среду разработки Eclipse

Библиотека работает на Java Runtime Environment версии 6 и выше в связи с активным использованием возможностей платформы Java версии 6. Исходный код библиотеки опубликован в интернете [9] под открытой лицензией по адресу `http://bitbucket.org/amelentev/juast/` . Библиотеку планируется опробовать в рамках реального техпроцесса компаний, занимающихся разработкой промышленного ПО.

# 4. Реализация

Несколько слов о реализации обсуждаемой библиотеки. Для интеграции с компилятором библиотека использует стандарт JSR269: Pluggable Annotation Processing [5]. JSR269 изначально предназначен для обработки аннотаций и генерации исходного кода, но в данном проекте он используется для получения управления и доступа к синтаксическому дереву разбора.

Для каждого компилятора библиотека регистрирует свой обработчик аннотаций через ServiceLoader [10]. Обработчики получают управление от компилятора после этапа синтаксического анализа. Далее подходящий используемому компилятору обработчик аннотаций получает синтаксическое дерево. Но, так как семантический анализ ещё не пройден, в дереве отсутствует много важной информации, например о типах используемых переменных и методов. Поэтому обработчик встраивается в компилятор через внутренние интерфейсы и получает управление уже после семантического анализа, при котором в синтаксическое дерево заносится необходимая информация. Далее обработчик преобразует синтаксическое дерево конкретного компилятора в унифицированное синтаксическое дерево и передаёт его выбранным верификаторам.

Унификация синтаксического дерева достигается оборачиванием (wrapping) конкретных классов узлов дерева для приведения к общим интерфейсам. Таким образом, для каждого класса узла синтаксического дерева конкретного компилятора библиотека содержит класс-обёртку в соответствующий интерфейс унифицированного дерева. Эти классы-обёртки при обращении к своим детям возвращают соответствующие обёртки для них.

Запускаемые верификаторы выбираются, по умолчанию, через ServiceLoader или через параметр JSR269 "`-Ajuast.process`". При верификации для указания ошибок библиотека также предоставляет унифицированные интерфейсы и адаптеры к соответствующим реализациям компиляторов, с помощью которых можно указать место возникновения ошибки в синтаксическом дереве и поясняющий текст.

Библиотека имеет компонентную архитектуру, где зависимости между компонентами разрешаются через Dependency Injection с помощью библиотеки Google Guice 2. В адаптерах для трансформаций внутреннего AST компилятора в унифицированное AST применяется функциональный стиль с использованием библиотеки Functional Java. Библиотека разбита на несколько модулей и собирается с помощью системы сборки Apache Maven 2. Каждая часть библиотеки может быть использована отдельно. Зависимости разрешаются с помощью maven.

# 5. Планы

Проект находится в активной разработке. Дальнейшее развитие проекта предполагается в трёх направлениях. Прежде всего, предполагается апробация технологии в реальном процессе разработки промышленного ПО. Параллельно предполагается обеспечить интеграцию технологии в среду разработки IntelliJ Idea, которая использует свой собственный Java фронтенд для различных верификаций. Также

предполагается дополнить интеграцию в среды Eclipse и Netbeans интерфейсами конфигурации. И наконец, важной и нетривиальной задачей на пути получения удобной технологии является разработка языка описания верификаторов (либо путём адаптации какого-либо подходящего языка, либо путём создания собственного языка описания семантических правил).

## Список литературы

1. PMD, java source code verificator. [Электрон. ресурс]. Режим доступа: http://pmd.sourceforge.net/.

2. Findbugs, java byte code verificator. [Электрон. ресурс]. Режим доступа: http://findbugs.sourceforge.net/.

3. Ayewah N., Pugh W., Morgenthaler J. D., Penix J., and Zhou Y. Using findbugs on production software // *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, (New York, NY, USA). ACM, 2007. P. 805–806.

4. Klaus H. and Grigore R. An overview of the runtime verification tool Java PathExplorer // *Formal Methods in System Design*. 2004. Vol. 24, No. 2. P. 189–215.

5. JSR269: Pluggable Annotation Processing API. [Электрон. ресурс]. Режим доступа: http://jcp.org/en/jsr/detail?id=269.

6. Project lombok. [Электрон. ресурс]. Режим доступа: http://projectlombok.org/.

7. Papi M. M., Ali M., Correa Jr. T. L., Perkins J. H., and Ernst M. D. , Practical pluggable types for Java // *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, (Seattle, WA, USA). 2008. July 22–24. P. 201–212.

8. Universal AST project for Checker Framework. [Электрон. ресурс]. Режим доступа: http://code.google.com/p/checker-framework/wiki/Ideas#Universal_AST.

9. Java Unified Abstract Syntax Tree project. [Электрон. ресурс]. Режим доступа: http://bitbucket.org/amelentev/juast/.

10. java.util.ServiceLoader. [Электрон. ресурс]. Режим доступа: http://java.sun.com/javase/6/docs/api/java/util/ServiceLoader.html.

# Integration of semantic verification into Java compilers

Klepinin A. V. , Melentyev A. A.

**Keywords:** program verification, AST, Java, compiler, JSR269, javac, ecj, Eclipse IDE, Netbeans IDE.

This paper introduces a method for static semantic analysis of source codes at compilation time directly within standard compilers. The method is implemented via unified integration with Java compilers to get the full access to Abstract Syntax Tree (AST) of compiled files after the semantic analysis stage of compilation process. The unified integration is implemented by common AST interfaces and adapters to AST implementations of Sun/Oracle javac and Eclipse Compiler for Java (ecj) compilers. This method provides transparent integration with Eclipse and Netbeans IDE without a need for any special plugins. Several examples of program verification rules are presented to demonstrate the method.

**Сведения об авторах:**
**Клепинин Александр Владимирович**,
Уральский Государственный Университет им. А.М. Горького,
кандидат физико-математических наук, доцент;
**Мелентьев Артём Алексеевич**,
Уральский Государственный Университет им. А.М. Горького,
аспирант, научный сотрудник НИИ ФПМ.