ISSN 0146-4116, Automatic Control and Computer Sciences, 2011, Vol. 45, No. 7, pp. 408–412. © Allerton Press, Inc., 2011.
Original Russian Text © A.V. Klepinin, A.A. Melentyev, 2010, published in Modelirovanie i Analiz Informatsionnykh Sistem, 2010, No. 4, pp. 70–77.

Integration of Semantic Verifiers into Java Language Compilers

A. V. Klepinin and A. A. Melentyev

A.M. Gorky Urals State University, Russia

e-mail: alklepin@mail.itnet.ru, amelentev@gmail.com

Received October 14, 2010

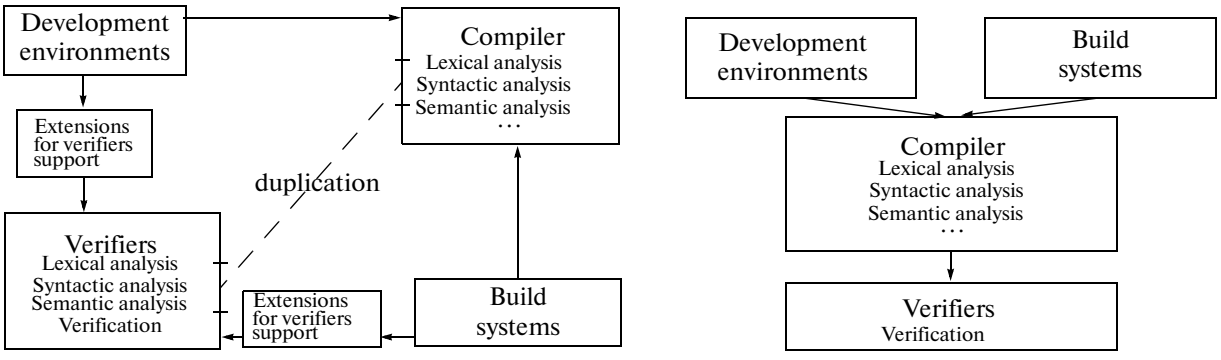
Abstract—This paper introduces one method for source code static semantic analysis at compilation time directly within standard compilers. The method is implemented via unified integration with Java compilers to get full access to Abstract Syntax Tree (AST) of compiled files after semantic analysis stage 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 integrated developer environments without a need for any special plugins. Several examples of program verification rules are presented to demonstrate the method.

DOI: 10.3103/S014641161107008X

1. INTRODUCTION

During the process of software development, universal programming languages are usually used. However, for some specific tasks, it is often appropriate to narrow the possibilities of the applicability of certain language constructions, because otherwise they can lead to guaranteed mistakes. It would be convenient to have an opportunity to perform additional semantic control of programs and reveal potentially dangerous or prohibited constructions in the current project.

Different external tools for verification of programs correctness (external static verifiers; for example, for the Java language, they are pmd [1] and findbugs [2, 3]) are often applied for general checks. However, for verification of specific domains, custom verifiers have to be written. Also, it can be mentioned (see Fig. 1a) that the first stages of the work of external verifiers based on the analysis of the source code (lexical, syntactic, and shallow semantic analyses) are the same as in the compiler. Why should they be started again if the work results of the compiler can be used? Besides, for the convenient application of external verifiers, their integration with different development environments and build systems must be provided. With verifiers embedded into the compiler, it will not cause any trouble any more. Both development environments and build systems will automatically start verification through the compiler without the necessity of installing special plugins (see Fig. 1b).



(a) external (b) embedded into a compiler

Fig. 1. Verifiers.

Here, the human factor should also be mentioned. Embedded tools of error diagnostics require immediate reaction from a programmer, while external tools embedded into the process of the product assembly reveal problems at the later stages, which results in the growth of the time spent for the problem's solution. Developers should see errors immediately in the process of the development without waiting for the long build process to be finished or manually starting external verifiers.

Errors can also be revealed during the program's execution [4] (dynamic verification), but this process is much more complex and consuming than error revealing during the compilation. Firstly, a special debugging assembly with embedded checks during the execution is required (thus, there exists a possibility that an error will be revealed in the released version, not in the debugging one). Secondly, a separate test launch of the program for error revealing is required (and there is no guarantee that it will find all the errors; some ways of the program's execution may not be considered as opposed to static analysis, which checks all the possible variants). It should be mentioned that sets of errors revealed by static and dynamic verification are incomparable as sets; i.e., it is not advisable to use only one type of verification during development. Therefore, the development of a convenient process of static verification combined with the compilation process seems to be a reasonable and actual goal.

2. TASK ASSIGNMENT

One of the way of providing a semantic analysis at the stage of compilation is integration with the compiler with the purpose of providing access for the verifier to the AST formed during the program's compilation process. Usually, there exist several versions of compilers for one language. For example, classical javac, which is a part of the Java Developer Kit and used in the development environment called Netbeans IDE, and the Eclipse Compiler for Java (ECJ), which is used in the development environment called Eclipse IDE, are the most well-known compilers for the Java language. Often, absolutely different algorithms are used in different compilers and, correspondingly, the syntax trees are. It would be convenient if verifiers worked with any compilers. In the Eclipse and Netbeans IDE cases, it would automatically mean that messages about problems in the code would appear directly in the development environment with references to the places of their appearance and other information during the compilation process without launching of any external utilities.

JSR269 Pluggable Annotation Processing [5] standard on Java compilers defines interfaces that provide some access for reading information from AST of compiled programs, but these interfaces reflect only the basic structure of the program and do not provide information about its implementation. Therefore, JSR269 of itself is useless for verification needs.

Although there exists integration into Java compilers in the Lombok library [6], the problem of compatibility of compilers is not solved there yet. For integration of special Lombok handlers (extenders, verifiers) with each specific compiler, a separate code that uses a separate family of classes bound to the compiler must be written. Besides, the project is oriented for the expansion of the Java language abilities (through the AST modification) via annotations. Lombok uses JSR269 for acquiring control during compilation. In the future, Lombok versions with unification of ASTs of compilers are planned, so that it will be possible to work with one unified family of classes that is not bound to compilers.

Similar integration also exists in the tool of the static verification Checker Framework [7], but integration there is implemented only for version 7 of the javac compiler. Besides, the Checker Framework is bounded only with type checking with the usage of annotations (*NotNull*, *Immutable*, *ReadOnly*, and others). The Checker Framework also plans to use the unified AST in the future versions [8]. Many other unifiers can be adapted to the unified AST, and, therefore, their area of applicability can be significantly expanded.

The goal of this work is designing a tool of program verification combined with the process of compilation for different compilers and development environments for the Java language. Designing a unified analysis tree, which is necessary for this goal's implementation, is important, actual, and useful of itself for many verification tools.

3. RESULTS

As a result of the performed work, the library juast [9] with an open source code has been created. Using this library, one can put semantic constraints on the code processed by the compiler as well as modify the AST of the program, therefore realizing the possibility of the programming language extension. Acquired verifiers and extensions will work everywhere where javac and ecj compilers are used, including such

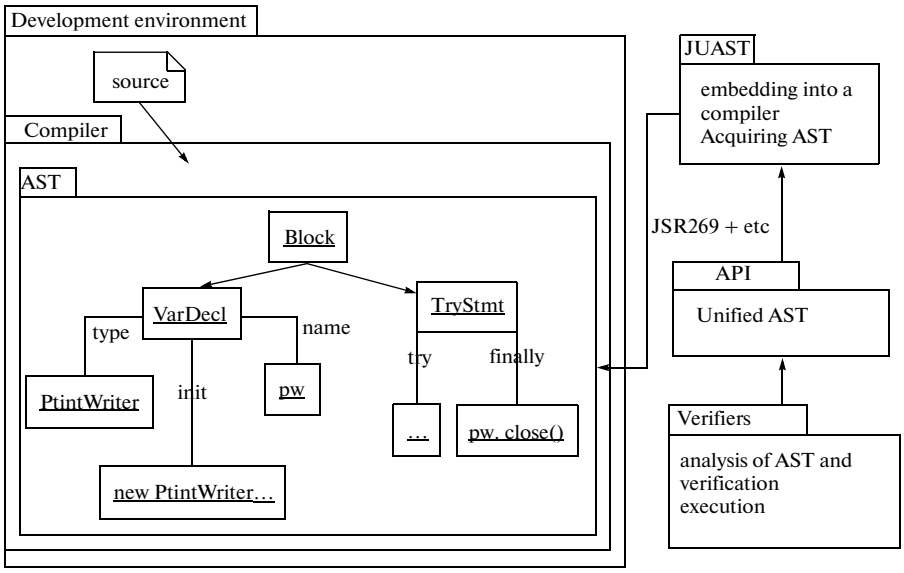


Fig. 2. Scheme of access to a compiler.

development environments as Eclipse IDE and Netbeans IDE, the Ant and Maven assembly systems, and other tools.

Juast connects to compilers and build systems as an ordinary library (e.g. javac –cp juast.jar Test.Java) without any modification of the compiler and provides unified access to the syntax trees of compiled programs after the stage of semantic analysis (see Fig. 2).

Integration into compilers and development environments is implemented through JSR269 interfaces [5] and some undocumented methods, as there is no direct and public access to the analysis trees in compilers.

In order to provide unification of the work with the javac and ecj compilers (and, correspondingly, the Netbeans and Eclipse environments), interfaces of the unified syntactic tree (AST) of the Java language and adapters for the compilers are implemented in the library. The AST interfaces provide access to the data about ancestors (it should be noted that there is no such information in javac and ecj trees) and implements the standard programming pattern "visitor" for AST traversal.

During the process of the AST traversal, verifiers can check the program's correctness and mark the revealed errors in the tree nodes. Compilers and development environments show such error messages in their standard view with references to the places of their appearance. Then, if critical errors have not been found, the standard process of compilation (optimization and code generation) continues.

Some simple verifiers that illustrate the possibilities of the technology are implemented on basis of the juast library. For example, a check that the external resources used by the program are correctly closed in the "finally" section of the corresponding "try" block (see Fig. 3) is implemented there. Also an overload of "+" "–" "*" "/" operators for BigInteger and BigDecamical types is implemented for testing the extension possibilities of compilers. In principle, this approach can be generalized to full operator overload support.

Considering the active using of the features of the Java platform version 6, the library works on the Java Runtime Environment version 6 and above. The source code of the library is published on the Internet [9] under an open license at http://bibucket.org/amelentev/juast/. The library is planned to be tested in the real technical processes of companies that develop industrial software.

4. IMPLEMENTATION

Let us say some words about the implementation of the discussed library. For integration with the compiler, the library uses the JSR269 Pluggable Annotation Processing [5] standard. JSR269 was initially intended for annotation handling and source code generation, but, in this project, it is used for acquiring control and access to the syntactic analysis tree.

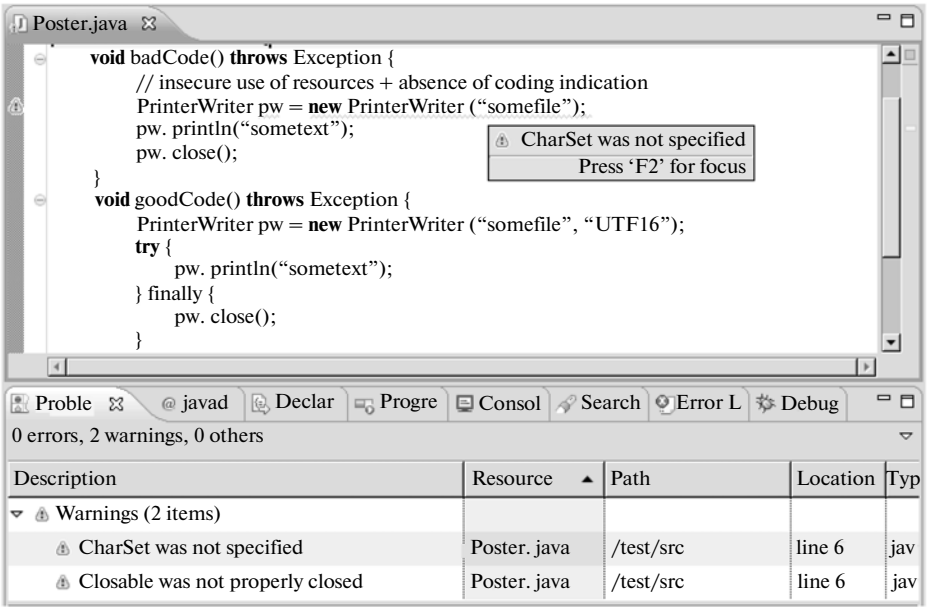


Fig. 3. The effect of the juast.jar library on the Eclipse development environment.

For each compiler, the library registers its own annotation handler through the ServiceLoader [10]. Handlers get control from the compiler after the stage of the syntactic analysis. Then, the annotation handler that corresponds to the compiler being used gets the AST. However, as the semantic analysis is not yet complete, the tree lacks a lot of important data, e.g., about the types of variables used and the methods. That is why the handler embeds into the compiler through the internal interfaces and gets control after the semantic analysis, during which necessary information is being put into the tree. Then, the handler transforms the syntactic tree of the specific compiler into the unified syntactic tree and passes it to the chosen verifiers.

Unification of the AST is achieved by wrapping of concrete classes of tree nodes with the purpose of bringing them to common interfaces. Thus, for each node class of the AST of the specific compiler, the library contains a wrap class for the corresponding interface of the unified tree. Addressing to their children, these wrap classes return corresponding wraps for them.

By default, the verifiers launched are chosen through the ServiceLoader or JSR269 parameter "-Ajuast.process." With the purpose of error reporting, during verification, the library also provides unified interfaces and adapters for the corresponding implementations of the compilers, using which the place of the error appearance in the AST and the illustrating text can be provided.

The library has component architecture, in which the dependences between the components are solved through Dependency Injection with help of the Google Guice 2 library. The functional style with the use of the Functional Java library is applied in the adapters for the internal AST compiler transformation into the unified AST. The library is divided into several modules and is assembled with the help of the Apache Maven 2 build system. Each part of the library can be used separately. The dependences are solved with the help of maven.

5. PLANS

The project is at the stage of active development. The following development of the project is supposed to take place in three directions. First of all, the approbation of the technology in the real process of industrial software development is supposed. Simultaneously, it is supposed to provide integration of the technology into the IntelliJ Idea development environment, which uses its own Java front end for different verifications. Also, it is supposed to supplement integration into the Eclipse and Netbeans environments with configuration interfaces. Finally, an important and unconventional task on the way of acquiring conve-

nient technology is the one of development of the verifier description language (either by the adaptation of some appropriate language or by the creation a language of the description of the semantic rules).

REFERENCES

1. *PMD, Java Source Code Verificator*. http://pmd.sourceforge.net/
2. *Findbugs, Java Bute Code Verificator*. http://findbugs.sourceforge.net/
3. Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., and Zhou, Y., Using Findbugs on Production Software, *Proc. 22nd ACM SIGPLAN Conf. on Object-Oriented Programming Systems and Applications Companion, OOPSLA'07*, New York, 2007, pp. 805–806.
4. Klaus, H. and Grigore, R., An Overview of the Runtime Verification Tool Java Path Explorer, *Formal Methods in System Design*, 2004, vol. 24, no. 2, pp. 189–215.
5. *JSR269: Pluggable Annotation Processing API*. http://jcp.org/en/jsr/detail?id=269
6. *Project Lombok*. http://projectlombok.org/
7. Papi, M.M., Ali, M., Correa, T.L., Jr., Perkins, J.H., and Ernst, M.D., Practical Pluggable Types for Java, *Proc. 2008 Int. Conf. on Software Testing and Analysis, ISSTA 2008*, Seattle, USA, 2008, pp. 201–212.
8. *Universal AST Project for Checker Framework*. http://code.google.com/p/checker-framework/wiki/ideas#universal_AST
9. *Java Unified Abstract Syntax Tree Project*. http//bitbucket.org/amelentev/juast/
10. *java.util.Service Loader*. http://java.sun.com/javase/6/docs/api/java/util/serviceloader.html