A JML Compiler on Eclipse Platform

[Project Proposal]

Amritam Sarcar
Dept. Computer Science, University of Texas, El Paso
500 West University Avenue
El Paso, TX - 79968
1-325-513-5214
amritamsarcar@yahoo.co.in

ABSTRACT

Java Modeling Language tools cover the full range of verification from runtime assertion checking (RAC) to full static program verification, with extended static checking (ESC) in between. Unfortunately, developers trying to do this must use separate applications and deal with problems like the tools accepting slightly different and incompatible variants of JML. Tool consolidation has become vital. This paper presents the architecture and design rationale behind a JML Compiler on Eclipse Platform, an extension of [1]., with a more detailed and focused on runtime assertion checking

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*programming by contract, assertion checkers, class invariants*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, invariants, assertions*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*object oriented programming*; D.3.2 [**Programming Languages**]: Language Classifications—*JML.*

General Terms

Languages

Keyword

JML, run-time checking, design by contract, interface violation.

1. INTRODUCTION

The Java Modeling Language (JML) is the most popular Behavioral Interface Specification Language (BISL) for Java. JML is recognized by a dozen tools and used by over two dozen institutions for teaching and/or research, mainly in the context of program verification [2]. Tools exist to support the full range of verification from runtime assertion checking (RAC) to full static program verification (FSPV) with extended static checking (ESC) in between [3]. Of these, RAC and ESC are the technologies which are most likely to be adopted by mainstream developers because of their ease of use and low learning curve.

Unfortunately, the current version, accept slightly different and incompatible variants of JML—sadly this is the case for practically all of the current JML tools. The reasons behind it are partly historical—

- the tools were developed independently, each having their own parsers, type checkers, etc. and
- partly due to the rapid pace of evolution of both JML and Java.

Not only does this last point make it difficult for individual research teams to keep apace, it also results in significant and unnecessary duplication of effort. For some time now the JML community has recognized that a consolidation effort with respect to its tool base is necessary. In response to this need, three prototypical “next generation” tools have taken shape: JML3, JML4, and JML5. This paper is a possible extension of JML4.

2. BACKGROUND AND GOALS

In this section we discuss the main goals to be satisfied in this project. Before doing so we give a brief summary of the runtime assertion checking

2.1 Runtime Assertion Checking

Assertions are formal facts about the state of a program; they are statements that are true at certain points in program code [4]. They are very useful for both debugging and proving correctness of programs [5]. There may be several ways to support assertions in programming languages, but one of the most popular approaches is to use macro statements that are expanded into appropriate program statements by preprocessors. The main examples are the assertion facilities of C [6] and C++[7] [8] (e.g., the assert macro).Meyer promoted simple assertions into what is referred to as the *design by contract* (*DBC*) [9][10]. There are various notations and tools that vary widely in their techniques and approaches to checking assertions at runtime from simple macro preprocessing and compiling to customized class loaders with the on-the-fly byte code manipulation.

2.2 Evolution of IDEs and Problem

2.2.1 Evolution of IDEs

With a phenomenal increase in the popularity of modern IDEs like Eclipse, it seems clear that to increase the likelihood of getting wide spread adoption of JML, it will be necessary to have its tools operate well within one or more popular IDEs.

2.2.2 Problem

Since we are targeting mainstream industrial software developers as our key end users, from an end user point of view, we strive to offer a single Integrated (Development and) Verification

Environment (IVE) within which they can use any desired combination of RAC, ESC, and FSPV technology. No single tool currently offers this capability for JML. One of the important challenges faced by the JML community its keeping up with the accelerated pace of the evolution of Java. There is little or no reward for developing and/or maintaining basic support for Java. While such support is essential, it is also very labor intensive.

2.3 Goals

Since JML is essentially a superset of Java, most JML tools will require, at a minimum, the capabilities of a Java compiler front end. Some tools (e.g., the RAC) would benefit from compiler back-end support as well. Hence, an ideal solution would be to extend a Java compiler, already integrated within a modern IDE, whose maintenance is assured by a developer base outside of the JML research community. If the extension points can be judiciously chosen and kept to a minimum then the extra effort caused by developing on top of a rapidly moving base can be minimized.

In summary, our general goals are to provide

- Propose an architecture for a new JML compiler on Eclipse platform.
- Extending the work in [1] for JML4 by implementing support for JML as extensions to the base support for Java so as to minimize the integration effort required when new versions of the IDE are released.
- This project would primarily be focused on runtime assertion checking on Eclipse platform.

3. PROPOSED ARCHITECTURE

Before proposing our new architecture we present the eclipse architecture.

3.1 Eclipse Architecture

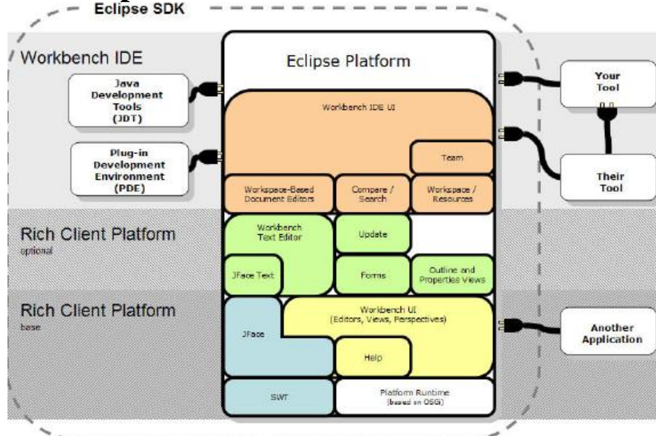

Figure 1. The Eclipse Architecture showing the Eclipse SDK and RCP

Eclipse is a plug-in based application platform. An Eclipse application consists of the Eclipse plug-in loader (Platform Runtime component), certain common plug-ins (such as those in the Eclipse Platform package) along with application specific plug-ins. Well known bundles of Eclipse plug-ins include the Eclipse Software Development Kit (SDK) and the Eclipse Rich Client Platform (RCP). While Eclipse is written in Java, it does not have built-in support for Java. Like all other Eclipse features, Java support is provided by a collection of plug-ins—called the Eclipse Java Development Tooling (JDT)—offering, among other things, a standard Java compiler and debugger.

The main packages of interest in the JDT are the ui, core, and debug. As can be gathered from the names, the core (non-UI) compiler functionality is defined in the core pack-age; UI elements and debugger infrastructure are provided by the components in the ui and debug packages, respectively.

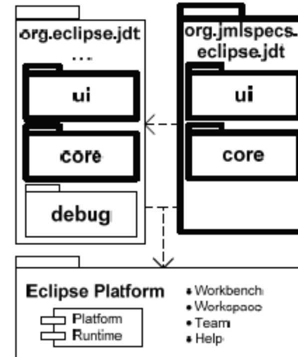

Figure 2. High-level package view

3.2 JML Compiler Architecture

At the top-most level, JML4 consists of customized versions of the org.eclipse.jdt.ui and org.eclipse.jdt.core packages (details will be given below) that are used as drop-in replacements for the official Eclipse JDT core and ui. These packages are shown in bold in Figure 2.

In the Eclipse JDT (and JML4), there are two types of parsing: in addition to a standard full parse, there is also a diet parse, which only gathers signature information and ignores method bodies. When a set of JML an-notated Java files is to be compiled, all are diet parsed to create (diet) ASTs containing initial type information, and the resulting type bindings are stored in the lookup environment (not shown). Then each compilation unit (CU) is fully parsed to fill in its methods' bodies. During the processing of each CU, types that are referenced but not yet in the lookup environment must have type bindings created for them. This is done by first searching for a binary (*.class) file or, if not found, a source (*.java) file. Bindings are created directly from a binary file, but a source file must be diet parsed and added to the list to be processed. In both cases the bindings are added to the lookup environment. If JML specifications for any CU or referenced type are contained in a separate external file (e.g. a *.jml file), then these specification files are diet parsed and the resulting information merged with the CU AST (or associated with the binding in the case of a binary file). Finally, flow analysis and code generation are performed. Our point of concern would be implementation of RAC using different existing methodologies.

4. APPROACH

We would informally describe here our approach towards achieving the goal, elicited above.

- Start with trying to understand the eclipse framework. This knowledge is essential because eclipse is a plug-in architecture. This would help us to customize the packages required for JML specifications.

• Exploring different existing ways to implement the runtime assertion checker on eclipse platform. They may include preprocessing[11], wrapper classes[12], direct byte code generation and aspect oriented programming.

The idea behind exploring different ways is to check that which way is apt for eclipse platform. We also require to know that a new version of eclipse would not have adverse affect on JML4 compiler architecture which would implement runtime assertion checking.

5. EVALUATION

We must admit that the evaluation criteria for verifying and validating that the proposed architecture indeed fits well into the eclipse platform has not been formally documented. However an informal description is given below.

• Choosing the extension points are most important. It should be judiciously chosen and kept to a minimum. This would facilitate minimizing the extra effort caused by developing on top of a rapidly moving base.

• One of the rules of Eclipse development is that public APIs must be maintained forever. This API stability helps avoid breaking client code. The following convention was established by Eclipse developers: only classes or interfaces that are not in a package named internal can be considered part of the public API. Hence, for example, the classes for the JDT's internal AST are found in the org.eclipse.jdt.internal.compiler.ast package, where as the public version of the AST is (partly) reproduced under org.eclipse.jdt.core.dom. Hence finding the hooks are very important for this project.

• The architecture so developed can be formally or informally analyzed using π-AAL[13]. Indeed, in addition to representing software architectures, we need to rigorously specify their required and desired architectural properties, in particular related to completeness, consistency and correctness. This would facilitate us in forming a formal, well-founded theoretically language based on the modal μ-calculus.

Other evaluation criteria may include performance issues, susceptibility whether the architecture so designed breaks (due to evolution of a new architecture of eclipse).

6. WORK FLOW

The manner in which we would try to achieve our goal is described below.

• Analyze the eclipse architecture.
• Understanding the eclipse framework.
• Explore the various ways of implementing runtime assertion checking as a standalone application and as well as on Eclipse platform.
• Conduct experiments and come up with an implementation of a prototype.

7. DELIVERABLES

The deliverables for this project would include one of the following. They are ordered in the ascending order of difficulty i.e. the most difficult is in the last.

• A report on implementing Runtime Assertion Checker on Eclipse. This report would include which methods were analyzed, evaluation criteria for choosing amongst the different methods.
• If such an architecture can be feasible, then we can formalize this new architecture using π-AAL.
• An implementation of the JML compiler on Eclipse platform.

8. CONCLUSION

In this paper, we have outlined a strategy for extending eclipse framework to incorporate JML. In particular use runtime assertion checking on eclipse platform.

This strategy is not without challenges, however. Two of the more troublesome are finding the right extension points and minimal change in the actual eclipse source code. The architecture that would be eventually chosen must adhere to certain specific criteria.

Acknowlegements

We gratefully acknowledge the support extended from Dr.Yoonsik Cheon, my research advisor who has always guided us whenever we were stuck with a problem.

9. REFERENCE

[1] Patrice Chalin, Perry R. James, and George Karabotsos. An Integrated Verification Environment for JML: Architecture and Early Results. *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 47-53, September 2007.

[2] G. T. Leavens, "The Java Modeling Language (JML)": http://www.jmlspecs.org, 2007.

[3] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications", International Journal on Software Tools for Technology Transfer (STTT), 7(3):212-232, 2005.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576{583, October 1969.

[5] Jurgen F.H. Winkler and Stefan Kauer. Proving assertions is also useful. *ACM SIGPLAN Notices*, 32(3):38{41, March 1997.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cli®s, N.J., 1978.

[7] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Co., Reading, Mass., 1990.

[8] Bjarne Stroustrup. *The C++ Programming Language: Third Edition*. Addison-Wesley Publishing Co., Reading, Mass., 1997.

[9] Bertrand Meyer. Applying \design by contract". *Computer*, 25(10):40{51, October 1992.

[10] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[11] Yoonsik Cheon. A Runtime Assertion Checker for the Java Modeling Language. Department of Computer Science, Iowa State University, TR #03-09, April 2003.

[12] R.P.Tan, S.H.Edwards. An Assertion Checking Wrapper Classes for Java. SAVCBS' 03 Helsinki, Finland.

[13] F. Oquendo. Formally modeling software architectures with the UML 2.0 profiles for π-ADL, *ACM SIGSOFT Software Engineering Notes,* 31(1):1-13, January 2006.