Runtime Assertion Checking Support for JML on Eclipse Platform
Amritam Sarcar
Department of Computer Science
University of Texas at El Paso,
500 W. University Avenue
El Paso TX 79968, USA
Email: asarcar@miners.utep.edu

Abstract
The Java Modeling Language (JML) is used to document design for Java and has been used as a common language for many research projects. The inability to support Java 5 features is reducing user base, feedback and the impact on JML usage. The performance of JML tools is also not that impressive. The JML/MAC compiler on average is five times slower than the Java compiler. In this paper, we present an architecture that would have better performance than JML2 and also try to alleviate the problem of extensibility of JML2 tools.

Java Modeling Language
The Java Modeling Language (JML) is a formal behavioral specification language for Java. It is used in detail design documentation of Java modules (classes and interfaces). JML has been extensively used by many researchers across various projects. JML has a large and varied spectrum of tool support. It extends from runtime assertion checking (RAC) to theorem proving.

Apart from all these tools, RAC and ESC/Java are most widely used amongst developers. However, lately there has been a problem for tool support. The problem lies in their ability to keep up with new features being introduced by Java. In this paper, we propose to redevelop JML compiler (jmlc) on top of a well maintained code base. We present the architecture that would support JML on an extensible architecture like Eclipse. We also present a new architecture for the JML/RAC compiler with potential performance gain than its predecessor.

Problems with JML Tools
• The Common JML tools, i.e., JML2 do not support robustness [1].
• The JML tools were built on an open source Java compiler and suffered from extensibility.
• The existing JML tools, more importantly RAC (the runtime assertion checker) from the performance point of view is very slow. The compilation time taken is huge compared to the compilation speed of Java (see Fig. 1).
• Three reasons can be cited immediately:
  1. Jmc does more work than Java.
  2. Jmc being built on an open source compiler, results in decreasing its performance. This compiler is not as efficient as Java.
  3. The compilation process of Jmc is double round. That is, every compilation unit undergoes two-time compilation.
• The third is the research question being addressed in this paper.

Double-round Architecture
• The normal flow of any Java source code starts from the scanner phase and ends in the code generation phase going through the different phases.
• For the case for JML-annotated Java source code, after the type checking phase, rather than going straight to the code generation phase, it goes for second-round of compilation.
• The major bottleneck for this architecture is the double-round compilation. This is because it affects the runtime performance.
• It is a well-known fact that in a compilation phase, most time is spent in the scanning phase (see Fig. 2).
• In this architecture, scanning and parsing is done twice for the original code which slows down the performance.

Figure 1. Relative-slowness of jmc compared to Java. Twenty-five programs that were test run for checking the compilation time were taken from the programs that were distributed as a part of the JML package, under the samples folder.

Figure 2. The average percentage of each phase on running twenty-five test cases. Those were taken from the programs that were distributed with JML package. They were timed on Eclipse platform.

Incremental Architecture
• The architectural style that we call incremental architecture works on the same fashion as the double-round architecture.
• The code that is sent to the scanner phase for the second round of compilation is not the entire code but only runtime code. Generally speaking, this kind of architecture actually supports abstract syntax tree (AST) merging mechanism (see Fig. 3). That is to say, the portion of code that is sent for second round of compilation, results into an AST. This new AST needs to be merged with the original AST.
• We must also note that the Eclipse framework does not provide us with any API that we can take help of for this incremental approach. The unit of increment in Eclipse is a compilation unit. However, in our case the unit of increment is a sequence of Java statements. The idea behind this approach is incremental compilation.
• This model parses and type checks the original source code (before RAC generation) in the first cycle of compilation, and uses this type checked AST to further mutate with the RAC version. The steps involved to implement this technique are (see Fig. 4)
• The main advantage of this architectural style is that the computation time would be greatly reduced.
• The lack of support from existing compiler framework or implementation may pose a serious problem.

Current Status and Future Work
• We have outlined a strategy for extending the Eclipse framework to incorporate JML RAC compiler into it. This strategy is not without challenges, however. Choosing the right extension points with minimal changes in the existing source code are difficult.
• On successful completion of the prototype we would eventually go onto full blown development with Concordia University and Kanazawa University.

References

Presentation of this poster was made possible in part by NSF Grant CNS-0540582.