3rd. Annual Meeting for Computer Alliance for Hispanic Serving Institutions (CAHSI), Googleplex, Mountain View, California, USA, January 2009

Runtime Assertion Checking Support for JML on Eclipse Platform

Amritam Sarcar[i]

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) 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 JML2 tools is also not that impressive. The JMLRAC compiler on average is five times slower than the Javac 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.

1. Introduction

The Java Modeling Language (JML) is a formal behavioral specification language for Java. It is used for detail design documentation of Java modules (classes and interfaces). JML has been used extensively 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.

Amongst 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 JMLRAC compiler with potential performance gain than its predecessor.

2. Problems with JML Tools

The Common JML tools, a.k.a JML2 do not support robustness [1]. The Common JML tools were built on an open source Java compiler and suffered from extensibility; by extensibility we mean language and tool extensions. For example, no JML tool yet supports the several new features of Java 1.5, the most important is the introduction of generics. The code base of this open source compiler was not built to support extensibility, the maintenance of which has become extremely difficult.

Another pressing problem of the JML2 tools is its performance. The existing JML2 tools, more importantly Jmlc (the runtime assertion checker) from the performance point of view is really very slow. The compilation time taken is huge compared to the compilation speed of Javac (see Fig.1). However, it is evident that since Jmlc does more work than Javac, it would take more time. The question that is more important to us is what is causing this slowness. Three reasons can be cited immediately:

- Jmlc does more work than Javac.
- Jmlc being built on an open source compiler, results in decreasing its performance. This compiler is not as efficient as Javac.
- The compilation process of Jmlc is double round. That is, every compilation unit undergoes two-time compilation.

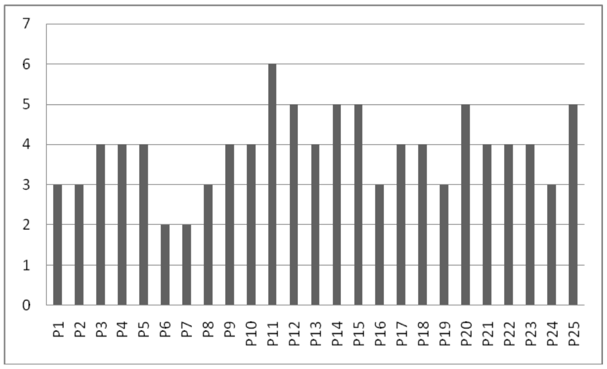


Figure 1. Relative-slowness of Jmlc compared to Javac. Twenty-five programs that were test run for checking the compilation time were taken from the programs that were distributed as a part of the JML package, under the samples folder.

Obviously there's nothing that we can do about the first. Regarding the second, there is work going on to build the next generation tools on the Eclipse platform [2], which is claimed to be more efficient. The third is the research question being addressed in this paper.

3. Double-round Architecture (Jml2 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. In this technique, the runtime code (which is in source code format) is directly merged into the original source code.

Addition of a special node, to depict that the node is a "special node"(for RAC purpose) is required for pretty printing [3]. On pretty printing, we fetch this new source code and resend to the scanning phase for the 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). Since this requires interacting with a slower device like hard-disk. In this architecture, scanning and parsing is done twice for the original code which slows down the performance.

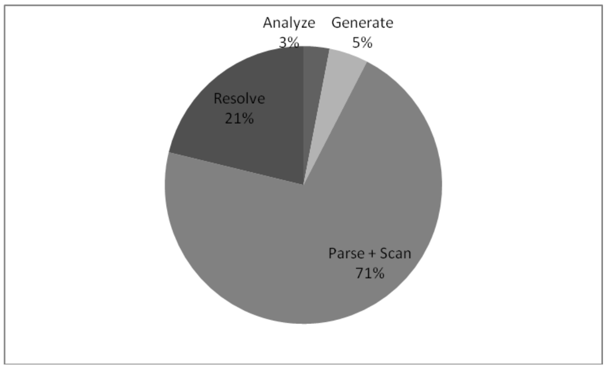


Figure 2. The average percentage of each phase on running twenty-five test cases. These were taken from the programs that were distributed with JML package. They were timed on Eclipse platform.

4. Incremental Architecture

The architectural style that we call incremental architecture works on the same fashion as the double-round architecture. However this time, 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.

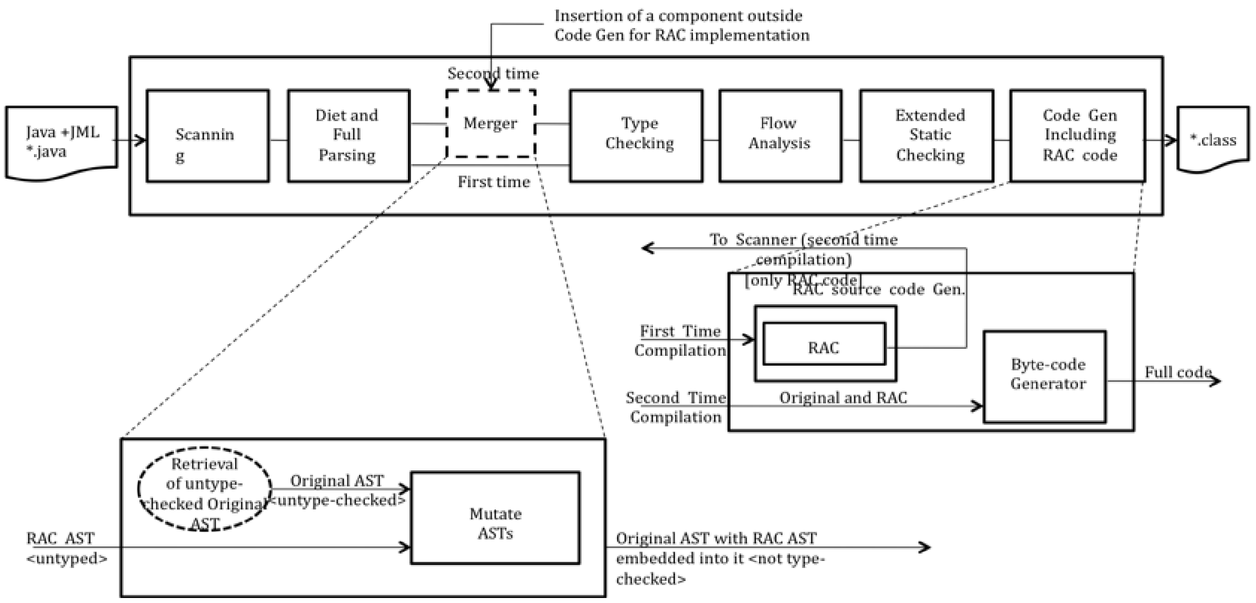
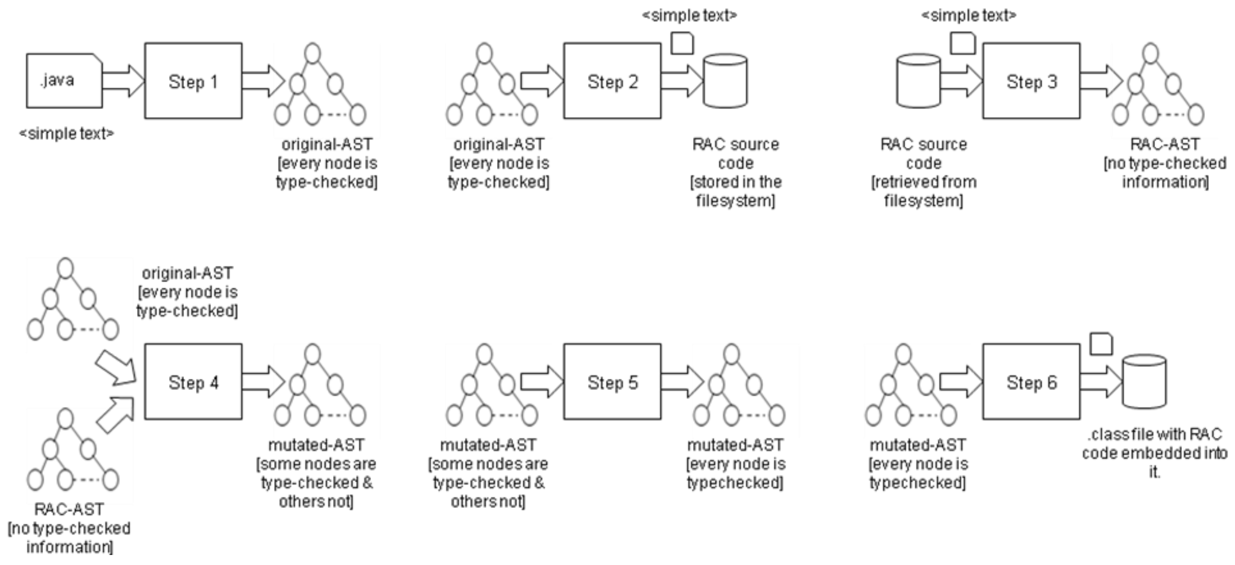

Insertion of a component outside Code Gen for RAC implementation
Java +JML *.java
Scannin g
Diet and Full Parsing
Second time
Merger
First time
Type Checking
Flow Analysis
Extended Static Checking
Code Gen Including RAC code
*.class
To Scanner (second time compilation) [only RAC code]
RAC source code Gen.
First Time Compilation
RAC
Second Time Compilation
Original and RAC
Byte-code Generator
Full code
Retrieval of untype-checked Original AST
Original AST <untype-checked>
Mutate ASTs
RAC AST <untyped>
Original AST with RAC AST embedded into it <not type-checked>

Figure 3. The incremental architecture designed on the Eclipse framework. Unlike in double-round compilation in this architecture only the RAC code is sent. In the second-round we merge using the RAC AST and the original AST to get the merged AST ready to go to byte-code generation phase.



<simple text>
.java
<simple text>
Step 1
original-AST [every node is type-checked]
original-AST [every node is type-checked]
Step 2
<simple text>
RAC source code [stored in the filesystem]
<simple text>
RAC source code [retrieved from filesystem]
Step 3
RAC-AST [no type-checked information]
original-AST [every node is type-checked]
RAC-AST [no type-checked information]
Step 4
mutated-AST [some nodes are type-checked & others not]
mutated-AST [some nodes are type-checked & others not]
Step 5
mutated-AST [every node is typechecked]
mutated-AST [every node is typechecked]
Step 6
.class file with RAC code embedded into it.

Figure 4. The status of the java source code and its intermediate format (AST) changes with every step. In step 1, in the first round the source code is changed to an AST, with the help of which RAC source code is generated and saved in a temporary folder. In step 3, this RAC code is retrieved and parsed and is merged with the original AST, which is done in step 4. In step 5 and 6 this merged AST is fully type checked and code generation is done.

We must also note that the Eclipse framework does not provide us with any API that we can take help of for this increment 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*. Since runtime assertion checking code generator basically generates valid Java statements on-the-fly, it should be possible for us to create an AST that would contain information of the runtime code only, and also be able to merge it with the original AST.

A key component of this interaction is the separation of the formation of AST nodes and binding them to their parent AST node. The complexity of this strategy is solely depended upon the following;

- Forming new AST nodes (in the second round, that of JML-specific statements). This AST must contain only runtime assertion checking information.
- Merging of the runtime AST with the original AST.
- Nullifying resolutions for generic types.

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)

1. In the first cycle, parse and type check the original source code.
2. Using this type checked AST, RAC code is generated in source code format, which is further saved in the temporary folder.
3. The RAC code is parsed; parsing the RAC code creates an initial AST.
4. This un-type checked AST (RAC-AST) is merged to the original type checked AST (original-AST).
5. Type-binding type-checking and flow-analysis is again done on this merged AST.
6. The resulting AST is sent for code generation.

The main advantage of this architectural style is that the computation time would be greatly reduced. The reason behind this is that even though, this approach does double-round compilation, for the second time, it only parses the newly added code which is the runtime code. However it also has some disadvantages. The lack of support from existing compiler framework or implementation may pose a serious problem. The original program code is changed by the preprocessor, i.e., line numbers of compiler errors do not actually fit the line numbers of the program. The same problem arises with debugging or runtime exceptions.

5. Current Status and Future Work

In this paper, 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. We are currently building the prototype that would support the features introduced in this paper.

On successful completion of the prototype we would eventually go onto full-blown development with Concordia University and Kansas University.

References

1. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An Overview of JML Tools and Applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212-232, June 2005.

2. 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.

3. Yoonsik Cheon. A Runtime Assertion Checker for the Java Modeling Language. *Technical Report 03-09* [The authors' PhD dissertation], Department of Computer Science, Iowa State University, Ames, Iowa, April 2003).

[i] The work of the author was supported in part by NSF under Grant No. CNS-0707874