A New Eclipse-Based JML Compiler Built Using AST Merging

Amritam Sarcar and Yoonsik Cheon

TR #10-08
March 2010

Keywords: incremental compilation; pre and postconditions; runtime assertion checking; AST merging; Eclipse; JML

1998 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; D.3.4 [*Programming Languages*] Processors — Compilers, incremental compilers; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

To appear in the *Second World Congress on Software Engineering*, December 19-20, 2010, Wuhan, China.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

A New Eclipse-Based JML Compiler Built Using AST Merging

Amritam Sarcar
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
amritams@microsoft.com

Yoonsik Cheon
Department of Computer Science
The University of Texas at El Paso
El Paso, TX 79968
ycheon@utep.edu

***Abstract*—The Java Modeling Language (JML) is a formal interface specification language to document the behavior of Java program modules and has been used in many research and industrial projects. However, its inability to support Java 5 features such as generics is reducing its user base significantly. Besides, the JML compiler is on average 8.5 times slower than the `javac` Java compiler. In this paper, we present a new JML compiler built on the Eclipse Java compiler to support Java 5 features. We used a technique called *AST merging* to implement coarse-grained incremental compilation. In our experiments we observed a significant improvement in compilation speed; the new compiler is 3 to 4.5 times faster than the current one.**

***Keywords*-incremental compilation; pre and postconditions; runtime assertion checking; AST merging; Eclipse; JML**

I. Introduction

The Java Modeling Language (JML) is a formal behavioral interface specification language for Java [1]. It is used for detail design documentation of Java program modules such as classes and interfaces. JML has been used extensively by many researchers and practitioners across various projects. It has a large and varied spectrum of tool support, extending from runtime assertion checking to theorem proving [2] [3]. Amongst these tools, the runtime assertion checking compiler, known as the JML compiler (`jmlc`) [4], is one of the most widely used tools by Java developers. However, there has been problems for tool support. One problem is its inability to keep up with new features such as generics being introduced by Java. Another problem especially with the JML compiler is its slow compilation speed. These problems are recognized as a barrier to a practical use of JML in an industrial setting [5].

In this paper, we present a new JML compiler built by extending the Eclipse Java compiler. The new JML compiler supports Java 5 features such as generics. Building the new compiler on a well-maintained, successful open-source code base will facilitate accommodating future language changes in Java easily. To address the runtime performance problem of the JML compiler, we introduced a technique called an *AST merging*. In essence, AST merging eliminates the need for parsing source code twice, as parsing (including scanning) is one of the most costly tasks during compilation; the `jmlc` compiler uses a double-round compilation scheme that parses a source code file twice (see Section IV for details). We observed that the AST merging technique is on average 1.4 times faster than the double-round strategy, and overall the new compiler is 3 to 4.5 times faster than the `jmlc` compiler. We also learned that the speedup increases as the code size increases, and thus we believe that new compiler will be better suitable for an industrial use.

The rest of this paper is organized as follows. In Section II below we give the necessary background information on JML, runtime assertion checking, and the Eclipse platform. In Section III we describe the problem of the current JML compiler focusing on its compilation speed. In Sections IV and V we explain in detail the designs of the current and the new JML compilers, respectively, by focusing on the double-round strategy and the AST merging. In Section VI we compare the two compilers through experiments, and in Section VII we review some of the most related work. In Section VIII we close this paper with a concluding remark.

II. Background

A. JML and Runtime Assertion Checking

JML is a formal behavioral interface specification language tailored for specifying both the runtime behavior and the syntactic interface of Java program modules [1]. It uses Hoare-style pre and postconditions to specify the behavior of a program module, and JML specifications are commonly written in a source code file as special comments. Figure 1 shows an example JML specification. The keyword **spec_public** indicates a private field, `courses`, is treated as public for specification purpose; for example, it can be used in the specification of a public method such as the `addCourse` method. As shown, a JML specification precedes the Java declaration such as a method declaration that it annotates. The **requires** clause specifies the precondition, and the **ensures** clause specifies the postcondition. In the postcondition, an old expression denotes the value of an expression in the pre-state. It is commonly used to specify the behavior of a mutation method.

JML can be used as a design-by-contract language for Java, as a significant subset of it can be translated to runtime checks by the JML compiler. For example, if the sample code is compiled with the JML compiler, every execution

```
public class CourseManager {
  private /*@ spec_public @*/ Set<Course> courses;

 /*@ requires !courses.contains(nc);
   @ ensures courses.contains(nc) &&
   @    (\forall Course c; \old(courses.contains(c));
   @      courses.contains(c)) &&
   @    (\forall Course c; courses.contains(c);
   @      \old(courses.contains(c)) || c == nc);
   @*/
  public void addCourse(Course nc) {
    courses.add(nc);
  }
  // other fields and methods
}
```

Figure 1. A Sample JML specification

of the `addCourse` method will be checked against its pre and postconditions. There are also tools based on the JML compiler that turn JML specifications into test oracles and perform automated testing [2].

B. Eclipse

Eclipse is a plug-in based application platform. All Eclipse features are supported through plug-ins. Java support is also provided through a collection of plug-ins, called the *Eclipse Java Development Tooling (JDT)*, that offers among other things a built-in standard Java compiler and debugger [6]. The Eclipse JDT code base is well maintained and relatively kept up to date with respect to language changes in Java.

One arching rule of Eclipse development is that public APIs must be maintained forever. This API stability helps avoid breaking client code. Because of this rule, however, there are two different packages concerned about abstract syntax trees (ASTs) for the Java programming language. The classes for the JDT internal ASTs are found in org.eclipse.jdt.internal.compiler.ast package, whereas the public version of the AST is partly reproduced in the org.eclipse.jdt.core.dom package.

III. Problems with the JML Compiler

The JML compiler (`jmlc`) is a key component of the Common JML tools available from the JML website (`http://www.jmlspecs.org`). It is one of the most widely used JML tools among Java developers. However, it has several problems including lack of support for Java 5 features, slow compilation speed, lack of integration with an IDE, and maintainability. In this section we focus on the compilation speed problem.

We performed an experiment to measure the compilation speed of the JML compiler, to compare it with that of a Java compiler, and to study the causes of its slowness. For the experiment we took sample Java programs included in the JML distribution (version 5.6RC4) available from the JML website. We used a total of 15 programs, and Table III shows the complexities of these programs in terms of the numbers of types, methods, fields, and source code lines. All programs are heavily annotated with JML specifications.

Table I
Complexity of sample programs

Program	Types	Methods	Fields	Lines
AlarmClock	4	17	11	389
Purse	3	8	6	192
Digraph	9	64	14	900
DirObserver	5	13	3	189
PriorityQueue	3	13	3	101
DLList	8	66	14	1228
TwoWayNode	8	70	10	1272
Counter	3	6	3	103
LinearSearch	4	14	1	221
Proof	1	4	2	241
Reader	4	11	11	257
SetInterface	3	23	7	782
BoundedStack	5	33	11	573
UnboundedStack	5	21	5	223
Entry	4	22	6	299

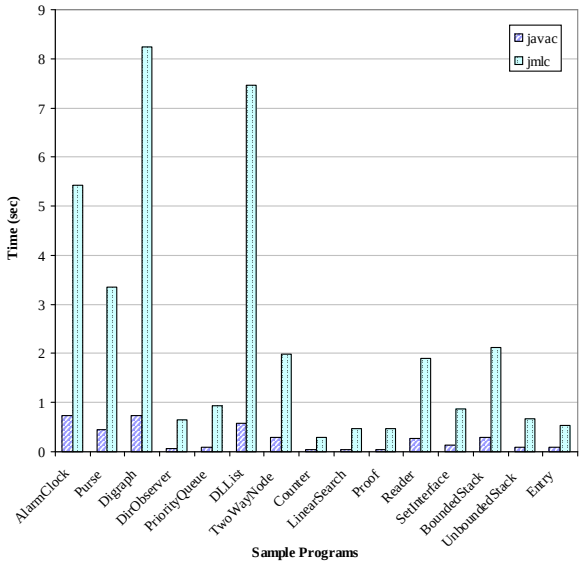

Figure 2. Compilation times of `jmlc` and `javac`

We compiled each of the sample programs with the JML compiler and the `javac` Java compiler (version 1.6.0.05) and measured its compilation time[1]. We used the Java hotspot Client VM version 1.6.0.05-b13. Figure 2 shows the result of our experiment. There is a huge difference in compilation time between `jmlc` and `javac`. The JML compiler is on average 8.5 times slower than the `javac` compiler.

$$r_{\text{avg}} = \frac{\sum_{i=1}^{n} r_i}{n} = \frac{\sum_{i=1}^{n} \frac{t_i^{\text{jmlc}}}{t_i^{\text{javac}}}}{n} \approx 8.5$$

We believe that several factors contribute to the slowness

[1]The experiment was performed on a Dell Inspiron I1420 Intel Pentium Dual CPU T2330@1.6 GHz with 2.00 GB RAM running Windows Vista Home Premium SP1.

of the JML compiler. First, the JML compiler has to do more work than a Java compiler because JML is a superset of Java. It is essentially a Java compiler with additional work to process and translate JML specifications to runtime checks. Second, the open-source Java compiler on which the JML compiler was built is not as efficient as the `javac` compiler[2]. Third, unlike a Java compiler, the JML compiler parses source code files of all directly or indirectly referenced types even if the corresponding, up-to-date bytecode files exist [4]. This is done to extract JML-specific type checking information such as **spec_public** because they are not currently encoded in bytecode in a readable form. In fact, our analysis indicates that this is on average responsible for about half of the slowness [7, page 22]. Finally, the JML compiler uses a double-round compilation strategy which requires source code to be processed twice [4] (see Section IV for details). There has been work to address the second factor by building new JML tools based on the Eclipse platform [5]. There is also work to encode JML specifications into bytecode. The last factor is the research question to be addressed in this paper.

IV. Double-round Compilation Strategy

Several approaches are possible for translating assertions such as JML annotations into executable code—for example, preprocessing, compilation, and bytecode manipulation. The JML compiler has the flavors of both preprocessing and compilation because it uses a double-round strategy [4] (see Figure 3). As said earlier, it extends an open-source Java compiler to process JML-specific declarations and annotations in source code. In particular, it introduces an additional compilation pass, "RAC code generation", to generate runtime assertion checking code after which the whole compilation passes are rewired to produce bytecode for both the original and assertion checking code at the same time. Note that the new pass is introduced after the type checking pass because certain type information is needed for the translation. The pass mutates the abstract syntax tree to add nodes for assertion checking code. If all the added nodes are in the type-checked form, then the compilation may proceed directly to the code generation pass, and this would be ideal in terms of compilation speed. However, the complexity of runtime assertion checking code makes it difficult to implement this. Thus, another compilation pass is introduced to pretty print the abstract syntax tree, containing both the original and the runtime assertion checking code, to a temporary file, which ends the first round of compilation. In the second round, the temporary file is compiled into bytecode by following the Java compilation passes.

It is a well-known that scanning is one of the most expensive phases during compilation because it has to interact

[2]The JML compiler (`jmlc`) was built by extending the MultiJava compiler (`mjc`) which was built on top of the Kopi Java compiler (`kjc`).

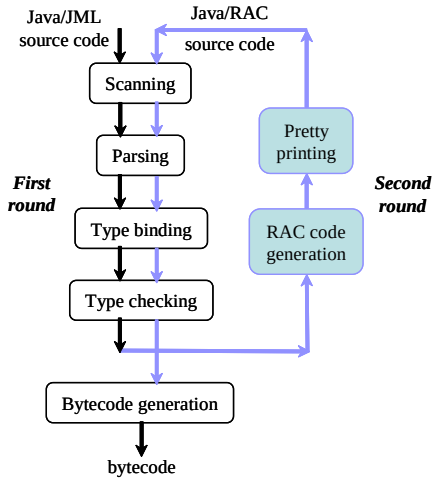

Figure 3. Double-round compilation strategy

with external devices and operates at the character level. For example, the Eclipse Java compiler spends 71% of the compilation time for scanning and parsing (see Figure 4). In the double-round strategy, scanning and parsing are performed twice for the original code, which we think is one of the reasons for the slow compilation speed of the JML compiler.

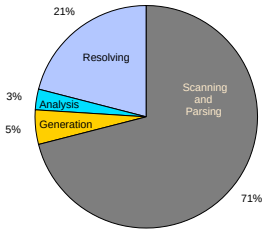

Figure 4. Distribution of compilation time

V. Incremental Compilation Using AST Merging

We propose an incremental compilation using a technique called an *AST merging* as a solution to the problem mentioned in the previous sections. Our approach works in the same principle as that of the double-round strategy in that it also consists of two compilation rounds. However, unlike the double-round strategy, only the instrumented runtime assertion checking code is sent to the second round of compilation for parsing. The key idea is to combine the abstract syntax trees (ASTs) of the original source code and the instrumented assertion checking code prior to code generation (see Figure 5). The main steps of our approach are summarized below.

1) Scan, parse, and type check the original source code including JML annotations.

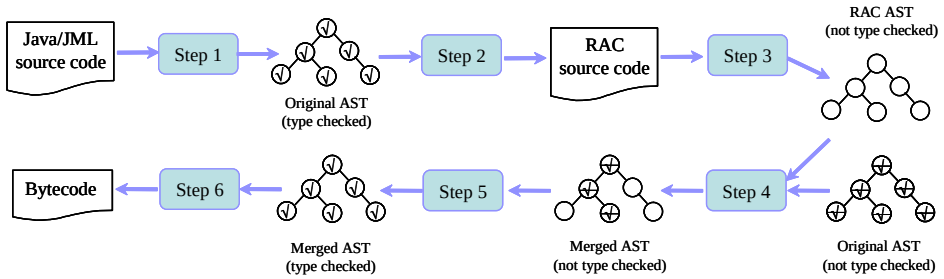

Figure 5. Main steps of AST merging: (1) produce AST of source code, (2) generate RAC code, (3) parse RAC code, (4) merge RAC and original ASTs, (5) type check merged AST, and (5) generate bytecode.

2) Generate runtime assertion checking (RAC) code from the type checked AST of the first step. The RAC source code is stored in an internal temporary file.
3) Parse the RAC source code stored in the temporary file to create its AST, called a RAC code AST.
4) Merge the RAC code AST of the previous step into the AST of original source code of the first step to produce a merged AST.
5) Follow the normal compilation passes with the merged AST by performing type checking and flow analysis.
6) Generate bytecode from the resulting AST of Step 5.

A key component of our approach is AST merging that combines two ASTs. In essence, the AST merging weaves the instrumented runtime assertion checking code into the original code by manipulating AST nodes. It is performed at three different levels such as compilation unit level, type level, and method level to insert new types, fields, methods, and code blocks to appropriate places of the original code (refer to [7] for a detailed description of the AST merging algorithm).

We implemented a new JML compiler, `jml4c`, on the top of the Eclipse Java compiler by using the approach described above [7]. There were several challenges in realizing the AST merging technique on the Eclipse platform. One challenge was that the Eclipse framework doesn't provide APIs for incremental compilation. In Eclipse, the unit of increment is a compilation unit, i.e., a file. For the JML compiler, however, the unit of increment is a sequence of Java statements because a JML annotation can be translated to a block of Java code. Another complication was that, in Eclipse, type-checking always starts from the top level construct, i.e., a compilation unit. Eclipse uses the visitor design pattern to visit all nodes of an AST starting from the root node. Upon visiting a node, if the node is already type-checked, Eclipse moves to the next (e.g., sibling) node without trying to visit the node's children. For this, each node has a sequence of bits that we call *AST bits* that acts as a blue-print for the node representing such information as whether the node is type checked. In short, this means that a block of instrumented Java code cannot be simply

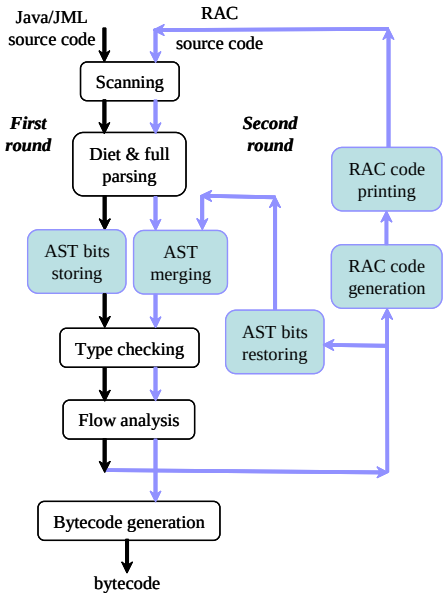

Figure 6. Incremental compilation using AST merging on Eclipse

parsed, type checked, and merged to the original AST. Remember that the original AST is already type checked, and the instrumented assertion checking code needs to be type checked in the context of the original AST as it will be merged to the original AST.

Figure 6 shows the architecture of the `jml4c` compiler. Scanning, diet and full parsing, type checking, flow analysis, and bytecode generation are the main compilation passes of the Eclipse Java compiler[3]. AST merging, AST bits storing, AST bits restoring, RAC code generation, and RAC code printing are new compilation passes introduced to support JML. As shown in the figure, Java compilation passes have two inputs, one for the first round of compilation and the other for the second round. The left arrow lines denote inputs for the first round compilation. As in the original

[3]In Eclipse, there are two types of parsing. Besides a full parsing, there is a diet parsing that only gathers signature information and ignores method bodies.

double-round approach, a Java file is scanned, parsed, type checked, and flow analyzed in the first round. However, prior to type checking, the AST bits of each node is backed up for later restoration (see below). Upon completion of type checking and flow analysis, the AST is now ready for RAC code generation. As before, RAC code is generated and then pretty printed. Note, however, that at this time only the RAC code—not woven in the original code—is pretty printed and sent for the second round of compilation; for this, a dummy context such as a compilation unit is created to host the instrumented RAC code. In the second round, the instrumented RAC code is scanned and parsed, producing a RAC code AST which is to be merged to the original AST. AST merging happens between parsing and type checking. The inputs to the AST merging pass are the original AST from the first round that is type checked and the RAC code AST from the second round that is not type checked. The AST merging involves two steps. First, the original AST is transformed to an untyped AST by restoring the AST bits of each node; remember that the AST bits were backed up prior to type checking. Second, two ASTs are merged by inserting RAC nodes to appropriate places of the original AST. Finally, the merged AST as a whole is type checked, flow analyzed, and then sent to the bytecode generation pass, which completes the compilation.

VI. Evaluation

We evaluated both our new JML compiler and the AST merging technique through a series of experiments. Below we summarize several measurements of the compilation speed.

In Section III we mentioned the problem of slow compilation speed of the current JML compiler, `jmlc`. Thus, we first compared the compilation speed of the new JML compiler, `jml4c`, against that of `jmlc`. For the comparison, we used the same set of sample Java code as in Section III. However, we used two sets of code, one with JML specifications and the other with all JML specifications removed by commenting them out. We measured the overall compilation times for each sample code by using two compilers, and Figure 7 shows the result. As shown, the new compiler performs better than the current compiler in both cases. According to our calculation, the `jml4c` compiler is on average about 3 times faster than `jmlc` in the presence of JML annotations and about 4.5 times faster than `jmlc` when all JML annotations are commented out.

In Section III we claimed that one possible cause of the slow compilation problem of `jmlc` is the double-round compilation approach. To learn about the gain from the use of AST merging, we compared the runtime performance of the AST merging technique against that of the double-round compilation approach and their impacts on the overall compilation speed. To make a fair comparison, we simulated double-round compilation on the Eclipse platform by slightly

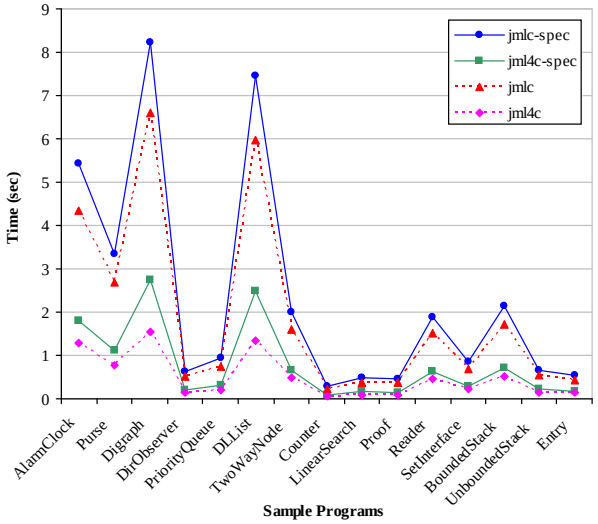

Figure 7. Compilation times of `jmlc` and `jml4c`

modifying our new compiler. We compiled each of sample JML files and measured its compilation time. Figure 8 shows the results along with the compilation times of `javac`. In all cases, the AST merging technique is faster than the double-round approach by a factor of 1.4. The graph also shows the efficiency the Eclipse Java compiler as base code; compare the compilation times of the double round approach on Eclipse with those of `jmlc` from Figure 2 in Section III. In a separate experiment, we also learned that the speedup increases as the size of source code increases. Thus, we think that the new compiler will perform better in practice, where a typical program may contain thousands of source code lines.

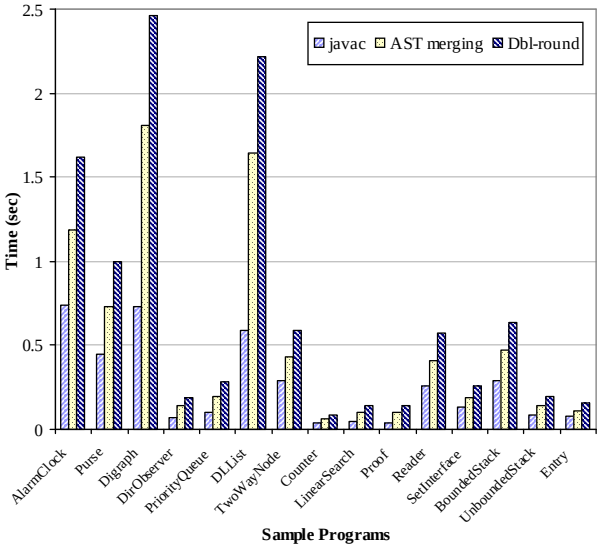

Figure 8. AST merging and double-round approach on Eclipse

We also measured the overhead of the new JML com-

piler compared to the Eclipse Java compiler. For this, we measured the compilation time of the sample code but with JML annotations commented out. On average, `jml4c` is 1.37 times slower than the Eclipse Java compiler.

VII. Related Work

There are many different notations and tools to support design-by-contract for Java [8]. The approaches vary widely from simple assertions to full contract enforcement tools. The implementation approaches also vary widely from preprocessing and compilation to bytecode manipulation, but preprocessing seems to be the most popular approach.

The notion of AST merging is not new. Other researchers have applied AST merging in implementing source code and program analysis tools. For example, Angyal et al. recently showed that AST differencing and merging techniques could be applied in model-driven software development [9]. In particular, synchronization between a platform independent model and a platform specific model was achieved using AST merging. However, most previous work focuses primarily on source code analysis and manipulation where both input and output are source code. In this paper, we explored a new direction of using AST merging in compilation for byte code generation.

Most incremental compilers has been built on systems that provide special data structures to store intermediate results and have tightly integrated components. For example, Fritzson demonstrated feasibility of incremental compilation at the statement level and also showed that to develop such a system extra information is required [10], which is not the case in a traditional batch compiler such as the Eclipse Java compiler. We showed in this paper that by using AST merging it is possible to develop a course-grained incremental compiler on top of a batch compiler without needing any special support from the underlying compiler framework.

VIII. Conclusion

We presented a new JML compiler, `jml4c`, developed by extending the Eclipse Java compiler. Besides supporting Java 5 features such as generics, we used the concept of AST merging to support coarse-grained incremental compilation. In essence, we eliminated the need for parsing source code twice—one for parsing the original source code and the other for parsing the original code woven with instrumented assertion checking code—by only parsing the assertion checking code and merging its AST to that of the original source code. Our experiments indicate that the new JML compiler is about 3 to 4.5 times faster than the current JML compiler. We also expect that the new JML compiler be able to easily incorporate future language changes in Java by using a well-maintained, open-source Java compiler as its base platform.

The `jm4c` compiler is open-source and available from http://www.cs.utep.edu/cheon/download/jml4c/.

Acknowledgment

The work of the authors was supported in part by NSF grants CNS-0509299 and CNS-0707874. The `jml4c` compiler was developed based on the JML front end—parser and type checker—developed by Patrice Chalin and his group at Concordia University.

References

[1] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary design of JML: A behavioral interface specification language for Java," *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 3, pp. 1–38, Mar. 2006.

[2] 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*, vol. 7, no. 3, pp. 212–232, Jun. 2005.

[3] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, "How the design of JML accommodates both runtime assertion checking and formal verification," *Science of Computer Programming*, vol. 55, no. 1-3, pp. 185–208, Mar. 2005.

[4] Y. Cheon, "A runtime assertion checker for the Java Modeling Language," Department of Computer Science, Iowa State University, Ames, IA, Tech. Rep. 03-09, Apr. 2003, author's Ph.D. dissertation.

[5] P. Chalin, P. R. James, and G. Karabotsos, "JML4: Towards an industrial grade IVE for Java and next generation research platform for JML," in *Verified Software: Theories, Tools, Experiments, Second International Conference, VSTTE 2008, Toronto, Canada, October 6-9, 2008. Proceedings*, ser. Lecture Notes in Computer Science, N. Shankar and J. Woodcock, Eds., vol. 5295. Springer, 2008, pp. 70–83.

[6] Eclipse, "Eclipse Java development tools (JDT)," 2010, retrieved on March 12, 2010 from Eclipse website: http://www.eclipse.org/jdt/.

[7] A. Sarcar, "Runtime assertion checking for JML on the Eclipse platform using AST merging approach," Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep. 10-01, Jan. 2010, author's MS thesis.

[8] L. Froihofer, G. Glos, J. Osrael, and K. M. Goeschka, "Overview and evaluation of constraint validation approaches in Java," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 313–322.

[9] L. Angyal, L. Lengyel, and H. Charaf, "A synchronizing technique for syntactic model-code round-trip engineering," in *15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, Mar. 2008, pp. 463–472.

[10] P. Fritzson, "Symbolic debugging through incremental compilation in an integrated environment," *Journal of Systems and Software*, vol. 3, no. 4, pp. 285–294, Dec. 1983.