JML: notations and tools supporting detailed design in Java

Gary T. Leavens, K. Rustan M. Leino, Erik Poll,
Clyde Ruby, and Bart Jacobs

TR #00-15
August 2000

Keywords: Behavioral interface specification language, detailed design notation, Java language, JML language, ESC/Java, LOOP.
1999 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, tools, JML, ESC/Java, LOOP; D.2.4 [*Software Engineering*] Software/Program Verification — Class invariants, correctness proofs, formal methods, programming by contract, reliability, tools, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, logics of programs, pre- and post-conditions, specification techniques;

To appear in OOPSLA 2000 Companion, Minneapolis, Minnesota, October 2000.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

JML: notations and tools supporting detailed design in Java

Gary T. Leavens*
Clyde Ruby
Iowa State University
Ames, Iowa, USA
{leavens,ruby}@cs.iastate.edu

K. Rustan M. Leino
Compaq Systems Research Center
Palo Alto, California, USA
rustan.leino@compaq.com

Erik Poll
Bart Jacobs
University of Nijmegen
Nijmegen, The Netherlands
{erikpoll,bart}@cs.kun.nl

ABSTRACT

JML is a notation for specifying the detailed design of Java classes and interfaces. JML's assertions are stated using a slight extension of Java's expression syntax. This should make it easy to use. Tools for JML aid in static analysis, verification, and run-time debugging of Java code.

Keywords

Behavioral interface specification language, detailed design notation, Java language, JML language, ESC/Java, LOOP.

1. INTRODUCTION

JML [8, 7], which stands for "Java Modeling Language," is useful for specifying the detailed design of Java classes and interfaces. It can be used to specify the details of the interface and behavior of such Java modules including pre- and postconditions for methods and class invariants.

JML is a cooperative, open effort. The notation and tools are currently being developed at Compaq Systems Research Center, the University of Nijmegen, and Iowa State University. However, we welcome participation by others who wish, for example, to extend it for new uses, perform case studies, develop tools, or work on semantic issues.

2. EXTENDED STATIC CHECKER

At Compaq Systems Research Center, the Extended Static Checker (ESC) project [9] uses a subset of JML as an annotation language for its ESC/Java tool. This tool can automatically check for certain kinds of common errors in Java code, such as dereferencing `null` or indexing an array outside its bounds. These checks are done statically and automatically; the checking is done without running the code and the user does not intervene other than to supply annotations in the code. ESC/Java takes the provided annotations into account to suppress spurious warning messages.

For example, in the following, the lightweight JML annotation that starts with `//@` contains a single precondition. This precondition says that the formal parameter, `descript`, may not be the `null` reference.

```
//@ requires descript != null;
public String deleteAtAfterNl(String descript)
{ /* ... */ }
```

The annotations provided also cause ESC/Java to perform additional checks. For example, ESC/Java would warn if an actual parameter to the above method could be `null`. Thus adding JML annotations helps give better quality warnings, use of ESC/Java fosters more annotations, and in turn these annotations help the tool do a better job of checking code for potential errors.

3. JML CHECKER

At Iowa State University, tools are being developed that use JML annotations in generating HTML documentation [12], and that use JML annotations for run-time debugging [1].

The HTML pages generated from JML annotated specifications are similar to those produced by Javadoc [3]. However, the documents also contain information from JML annotations, including method specifications, class invariants, etc.

The prototype tool that uses JML annotations for run-time debugging can automatically check preconditions of methods. It can also automatically check whether a class invariant holds at run-time. It does this by generating Java source code that contains the additional checks at the beginning of each method body. As in Eiffel [10], this capability helps in debugging code. Unlike Eiffel, the tool cannot yet check postconditions. However, the JML checker can handle annotations in Java interfaces (where no code can be directly added) and JML's model (specification-only) fields.

*The work of both Ruby and Leavens was supported in part by the US National Science Foundation under grant CCR-9803843.

To appear in *OOPSLA 2000 Companion* Minneapolis, Minnesota, October 2000.

Model fields are important for the specification of Java interfaces, and also allow more complete specification of collection classes than is easy to do in Eiffel.

Both of these tools handle JML's specification inheritance [2]. That is, they combine specifications from superclasses and superinterfaces and use those to form a complete specification of a class [13].

4. LOOP TOOL

The University of Nijmegen's LOOP tool [5] translates JML annotations into proof obligations. These proof obligations are used in an interactive theorem prover (PVS or Isabelle) to verify the correctness of a Java implementation. The translation from JML to formal proof obligations has required the Nijmegen group to formalize many details of Java and JML's semantics. Interactive theorem proving is labor-intensive, but allows verification of more complicated properties than can be handled by static checking. The two approaches (static and semantic checking) are complementary.

As a serious case study, the Nijmegen group is applying JML and the LOOP tool to Java smart cards [11]. The JAVA CARD language is a good target for this study, because it is a simple subset of Java that does not contain complex features as threads and dynamic loading.

5. THE JML NOTATION

JML is also interesting as a specification language.

JML blends the Eiffel [10] and Larch [4] traditions (and others which space precludes mentioning). Because JML supports model (specification-only) fields in classes, specifications can be more precise and complete than those typically given in Eiffel. However, because, like Eiffel, it uses Java's expression syntax in assertions, JML's notation is easier for programmers to learn than one based on the Larch Shared Language (LSL). On the other hand, to allow specifications to use mathematical theories like those that would be specified in LSL, JML includes several such theories, such as sets and sequences, but hides them behind a facade of Java classes. This allows these theories to be used in assertions as if they were a set of Java classes.

As in the Larch family, JML method specifications have a frame axiom (the `modifiable` clause), which says what fields a method may assign to. JML also allows various forms of redundancy in method specifications, including the statement of properties that are implied by the specification and examples that illustrate how a method can be used [4, 6, 14]. These can be used to call the reader's attention to various properties. A theorem prover could also use them as proof obligations, which could help debug the specification. The examples can also be used as method test cases.

6. REFERENCES

[1] A. Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, May 2000. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[2] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, Mar. 1996.

[3] L. Friendly. Design of Javadoc. International Workshop on Hypermedia Design '95, 1995.

[4] J. V. Guttag, J. J. Horning, et al. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, NY, 1993.

[5] B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 329–340. ACM, Oct. 1998.

[6] G. T. Leavens and A. L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In J. M. Wing et al., editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.

[7] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, Boston, 1999.

[8] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, Department of Computer Science, Feb. 2000. See `www.cs.iastate.edu/~leavens/JML.html`.

[9] K. R. M. Leino et al. Extended static checking. At `http://research.compaq.com/SRC/esc/Esc.html`, 2000.

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

[11] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In *Fourth Smart Card Research and Advanced Application Confernce (CARDIS)*. Kluwer Academic Publishers, 2000.

[12] A. D. Raghavan. Design of a JML documentation generator. Technical Report 00-12, Iowa State University, Department of Computer Science, July 2000.

[13] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report 00-03a, Iowa State University, Department of Computer Science, July 2000.

[14] Y. M. Tan. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering*. Kluwer Academic Publishers, Boston, 1995.