Demonstration of JML Tools

Gary T. Leavens, Yoonsik Cheon, and David R. Cok

TR #05-13
April 2005

Keywords: specification languages, runtime assertion checking, documentation, tools, formal methods, program verification, programming by contract, Java language, JML language.

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/ Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — computer-aided software engineering (CASE); D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, validation, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids, design, testing tools, theory; D.3.2 [*Programming Languages*] Language Classifications — Object-oriented languages; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques.

Submitted for publication

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Demonstration of JML Tools

Gary T. Leavens
Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, Iowa 50011-1041 USA
leavens@cs.iastate.edu

Yoonsik Cheon
Department of Computer Science, The University of Texas at El Paso
500 W. University Avenue, El Paso, Texas 79968-0518
cheon@cs.utep.edu

David R. Cok
Eastman Kodak Company, R&D Laboratories,
Rochester, New York, USA
cok@kodak.com

April 21, 2005

Abstract

The Java Modeling language (JML) is a behavioral interface specification language tailored to Java. This demonstration presents some of the basic tools for generating and browsing documentation, runtime assertion checking, and unit testing.

1 Introduction

This demonstration presents some fundamental tools that work with the Java Modeling language (JML) [10, 11]. While many groups provide various tools that work with JML [2], this demonstration focuses on the tools in the standard distribution of JML, which is freely available from `http://jmlspecs.org/`.

2 Background on JML

JML is a behavioral interface specification language tailored to the specification of Java classes and interfaces. To a first approximation JML is a cross between Eiffel [12] and the interface specification languages of the Larch family [9, 14]. Like Eiffel, JML supports design by contract by allowing invariants, and method pre- and post-conditions to be written using Java expressions. However, JML extends the set of expressions used in assertions with some extra operators, including quantifiers.

Like the Larch-style interface specification languages (and VDM), JML allows specifiers to describe abstract values of objects using a built-in mathematical library. This makes it easy to write fairly complete specifications of collection classes. However, unlike Larch-style interface specification languages, JML hides the mathematics behind a facade of Java classes. Also, unlike the Larch approach, JML allows specifiers to describe the abstract values of objects in several pieces, using several specification-only (model) fields.

3 The Tools to be Demonstrated

The demonstration will focus on three tools in the standard distribution of JML: the documentation generation tool, `jmldoc`, the runtime assertion checking compiler, `jmlc`, and the unit testing tool, `jmlunit`.

3.1 Jmldoc: The Documentation Generation Tool

The documentation generation tool, `jmldoc`, produces HTML web pages from JML specifications. In essence, it is a modified version of the `javadoc` tool [8] that understands JML specifications. The web pages produced by the `jmldoc` tool are similar to those produced by the `javadoc` tool. However they include the JML specifications as well as the usual informal English documentation.

Besides doing syntax and type checking of the JML specifications, one of the main features of the `jmldoc` tool is that its output shows the reader all of the inherited specifications that affect an interface, class, or method. In JML, public and protected non-static methods inherit the specifications of all methods that they override. This is helpful in that it shows implementors the obligations they have to fulfill due to specification inheritance.

3.2 Jmlc: The Runtime Assertion Checking Compiler

The runtime assertion checking compiler, `jmlc`, produces Java class files (bytecode) from JML-annotated source files [3, 4]. The JML annotations can appear either in the source itself or in a separate file. The compiled code checks all of the executable assertions when the code is run, and throws errors when assertion violations are encountered.

One of the most interesting aspects of the runtime assertion checking compiler is its support of JML's specification-only (model) variables. In the current `jmlc` references to such specification-only variables are realized by method calls. Such variables allow one to write assertions abstractly without referring to concrete program states, and are particularly useful for specifying container objects, such as collection classes [7].

Another interesting aspect of the runtime assertion checking compiler is its support for Java interfaces. Besides its ability to compile specification-only variables in interfaces, the tool can also handle other kinds of specification annotations in Java interfaces. In particular, `jmlc` can compile interfaces that include invariants and method pre- and post-conditions. Moreover, it incorporates the effects of multiple inheritance of specifications in such interfaces. This allows the specification of interfaces in class libraries, which describes common behavior that is inherited by many classes that implement the interface.

3.3 Jmlunit: The Unit Testing Tool

The unit testing tool, `jmlunit`, produces Java source files that can be used to test a class (or interface) [5, 6].[1] The unit testing tool works together with the JUnit unit testing framework [1]. The tool eases the process of testing by automatically producing a test oracle that decides test success or failure based on violations of runtime assertions. It automatically ignores test cases for a method m by ignoring those that cause m's precondition to be violated.

Using this tool, a user generates a class that will hold test data, and another class which serves as a test driver. After filling in some sample test data in the first class (done by hand, currently), tests can be run automatically. This makes testing easier and helps increase confidence in both code and specifications.

References

[1] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[2] Lilian Burdy, Yoonsik Cheon, David 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 (STTT)*, 2004. To appear.

[3] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation.

[4] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun, editors, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002*, pages 322–328. CSREA Press, June 2002.

[5] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In Boris Magnusson, editor,

[1] Parasoft's independently-developed Jtest tool [13] is similar. However, the Jtest tool uses a specification language that has less ability to write specifications abstractly, and is not able to write model-oriented specifications for interfaces.

ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Máalaga, Spain, Proceedings, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[6] Yoonsik Cheon and Gary T. Leavens. The JML and JUnit way of unit testing and its implementation. Technical Report 04-02a, Department of Computer Science, Iowa State University, April 2004. Submitted for publication.

[7] Yoonsik Cheon, Gary T. Leavens, Murali Sitaraman, and Stephen Edwards. Model variables: Cleanly supporting abstraction in design by contract. *Software—Practice and Experience*, 35(6):583–599, May 2005.

[8] Lisa Friendly. The design of distributed hyperlinked programming documentation. In S. Fraïssè, F. Garzotto, T. Isakowitz, J. Nanard, and M. Nanard, editors, *Proceedings of the International Workshop on Hypermedia Design (IWHD'95), Montpellier, France, 1–2 June 1995*, pages 151–173. Springer, 1995.

[9] J. V. Guttag and J. J. Horning. Preliminary report on the Larch Shared Language. Technical Report 307, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1983.

[10] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Iowa State University, Department of Computer Science, April 2005. See `www.jmlspecs.org`.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, and Joseph Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, April 2005.

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

[13] Parasoft Corporation. Automatic Java^{TM} software and component testing: Using Jtest to automate unit testing and coding standard enforcement. Available from `http://www.parasoft.com/jsp/products/tech_papers.jsp?product=Jtest`, as of Feb. 2003.

[14] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

A Description of the Presentation

The presentation would follow roughly the following outline, which is similar to the outline of the main paper itself.

A.1 Introduction and Overview

First we would give an introduction to the JML project as a whole, and the JML specification language. We would describe the different research groups working on JML in very brief terms. We would also give a sample specification to be used both to explain the specification language and in the rest of the presentation.

A.2 Jmldoc

Next we would present a brief look at how to run the `jmldoc` tool, and then focus on its output features. We would give a tour of a specification of a small hierarchy of classes and interfaces. We would focus on showing the effects of specification inheritance and the ease of browsing in various directions using hyperlinks.

A.3 Jmlc

Next we would present a brief look at how to run the `jmlc` tool, and then focus on how it can be used to detect various errors that we would seed into sample code that implements the specification described in the previous parts. Time permitting, we could also look at the compilation strategy used to compile specifications in interfaces.

In this step we will also briefly present the `jml` tool, which only does type checking and not compilation. It is useful because `jmlc` is fairly slow.

A.4 Jmlunit

Next we would present a brief look at how to run the `jmlunit` tool, and then focus on how it can be used to do a unit testing. We would take some of the implementations used in the previous section of the demonstration and perform unit testing on them to reveal additional errors in specifications and code. We would demonstrate how to supply test data for different kinds of types involved in these samples (immutable, cloneable, and immutable objects without clone methods).

A.5 Summary

After this there would be a brief summary and then time for questions.

B Tool Availability and Maturity

The tools described in this demonstration are open source and freely available (see the next section).

The tools have been through several releases, but are not completely industrial quality. They've been used in classes and by a small number of people. There are still several efficiency issues with the tools, and in particular the runtime assertion checking compiler is quite slow in compilation time.

C Web-Page for the Tool

The JML tools described in this paper are downloadable from the sourceforge page for JML: `http://sourceforge.net/projects/jmlspecs`. More information about JML is available from the project's home page, which is located at `http://jmlspecs.org/`.