

Formal Honors Thesis Proposal:
Integration of Java Generics Into The `jml`
Tool Within The Eclipse IDE

by
Adrian Kostrubiak

Submitted in partial fulfillment of Honors Requirements
For the Computer Science Major,
Dickinson College, 2008-09.

Professor Tim Wahls, Supervisor.

September 22, 2008.

1. Background Information

1.1. Java Generics

In Java, generics were added to the platform with the release of the Java 5 JDK. Since then, generics have been an indispensable tool in Java programs, allowing for methods to operate on variables of any class type while at the same time supporting compile time type checking. This ultimately results in fewer run time errors.

1.2. JML

The Java Modeling Language (JML) is a behavioral interface specification language, which is a mathematical notation that can be included in Java programs through the use of in code annotations, which reside in comments. It is used to formally specify the behavior of modules (such as a method) or sections of code in a Java program. As such, JML follows the design by contract ideal. That is to say, that any methods annotated with JML comments should be constrained by both the pre and post conditions specified in the JML, if the code is indeed correct and upholds its contract (as per the design by contract design pattern). The following is a trivial example of some JML code:

```
/*@ assignable nothing;                                1.1
    ensures \result == j + k; */
public int sumOfTwo(int j, int k) { return(j+k); }
```

In this example, the `assignable` clause tells us that this method does not assign any class variables. The `ensures` clause tells us that the result (that is, the return value) is going to be the sum of `j` and `k`. There is no `requires` clause, as many methods would have, as this `sumOfTwo` method has no preconditions to be satisfied. However, should one wish to

implement the `sumOfTwo` method with preconditions, it would be as easy as simply adding a `requires` clause into the JML code before the method, such as `requires j > 0 && k > 0;`. This would change the method's contract such that it would mandate that the method be called with only positive integers rather than any possible interger.

Aside from the checking of contracts, JML has a multitude of other uses as well. Amongst these other uses of JML are included code documentation, runtime assertion checking and the ability to create proofs of program correctness based on the JML specifications (Leavens et al., 2000).

1.3. jmlTool

The `jmlTool` was designed to turn the specifications laid out in JML into constraint programs. These constraint programs can then be executed as normal Java byte code utilizing the libraries of the Java Constraint Kit (JCK). Having an executable formal specification for an application simplifies the development process for a software engineer greatly. By being able to actually run the specification, the developer is able to confirm the meaning of the specification and validate whether or not any given software module fits the specification (Wahls and Krause, 2006).

1.4 jmlTool Within Eclipse

In its current version, the `jmlTool` has been ported into the Eclipse architecture (Eclipse being a very popular and extremely powerful Java based IDE) through the work of Professor Tim Wahls. Many other researchers are currently working on the JML4 project, which is the integration of JML into Eclipse, however Professor Tim Wahls is the only one to

be working with the `jmlc` tool. Within this context, when a program is run, the program itself is not actually being run; rather, the `jmlc` tool translates the JML annotations into a constraint program, and these constraint programs are run with the inclusion and within the framework of the JCK library (Abdennadher et al., 2002). This turns the JML, found within annotations, into executable code rather than code from a non-executable formal language. The executable code can then be run in its own right, and checked for accuracy. This ability for the developer to check the meaning of the specification against what was truly meant through this type of “hands on” technique not only make developing the specification easier, but greatly simplifies the very understanding of the specification.

2. Moving Forward With The `jmlc` Tool

Unfortunately, one of the current failings of JML is the lack of support for generics. As generics now play a huge role in the correct and safer implementation of any sort of collection (almost all programs use at least some sort of collection), this is a lacking feature in the implementation of JML. Although there are a few people working on implementing these features, many of the finer details of the implementation have not yet been determined. For my thesis, I plan to study and research the design of generics as they are implemented in Java. After deciding upon an appropriate implementation, including matters such as relevant design patterns, I plan on integrating generic types into the JML project. This will be taken a step further by the integration of such generic types into the `jmlc` tool. The implementation steps will require a fair amount of research in order to be certain that the implementation is in fact correct. While `jmlc` is a very strong tool in its current incarnation, the added benefits of

being able to deal with generics would allow it to be used in a more complete manner in industry. With such additions, JML could make the leap from a mere academic pursuit to being a very powerful and robust suite that developers around the world would be able to leverage in the implementation of the design by contract ideals in their programs.

Given that Professor Tim Wahls has had extensive experience with the JML4 language and that he was the one to implement the `jmlc` tool into an Eclipse based environment, I will be working on this project under his guidance.

3. References

1. B. Krause and T. Wahls. jmlc: A tool for executing JML specifications via constraint programming. In L. Brim, editor, Formal Methods for Industrial Critical Systems (FMICS '06), volume 4346 of Lecture Notes in Computer Science, pages 293 – 296. Springer-Verlag, August 2006.
2. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In OOPSLA 2000 Companion, Minneapolis, Minnesota, pages 105–106. ACM, Oct. 2000.
3. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan, M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 7(3):212-232, June 2005.
4. L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. A case study in JML-based software validation. In *The 19th IEEE International Conference on Automated Software Engineering (ASE 04)*, pages 294–297, September 2004.
5. S. Abdennadher, E. Krämer, M. Saft and M. Schmauss. JACK: A Java Constraint Kit. In *Electronic Notes For Theoretical Computer Science*, volume 64, pages 1-17. September 2002.