

Research Report:
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.
Professor Grant Braught, Reader.
Professor John MacCormick, Reader.

December 1, 2008.

1. Introduction

In the recent movement towards smarter and more efficient software development, many software engineers have been looking for an effective means to verify the software modules that they have written. One of the directions that this search has taken is in the area of formal specifications, mathematical notations describing what a module should do as opposed to how it should complete its task. Due to the popularity of Java and the lack of a formal specification language, the Java Modeling Language (JML) was created.

Numerous tools have been created to work with JML, including runtime assertion checking and general code documentation. Although JML and the accompanying suite of tools are a powerful toolset for a developer, a few large hurdles remain to be dealt with before JML can effectively be used in industry, including the implementation of generics, which were added to Java with the release of the Java 5 JDK.

To this end, I have been researching and working on the implementation of Java generics into the `jmlc` tool, a tool which is used for creating executable versions of the specifications. What follows is a report of my progress thus far with this task, prefaced by an introduction to the relevant technologies involved.

2. Background Information

2.1. Java Generics

In Java, generics were added to the platform with the release of the Java 5 JDK. Since this addition, generics have become an indispensable tool in Java programming,

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.

2.2. 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 that reside in comments. JML is used to formally specify the behavior of modules, such as a method, in a Java program (Leavens et al., 2008). As such, JML follows the ideal of design by contract. This 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 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;                               (2.1)
   ensures \result == j + k; */

public int sumOfTwo(int j, int k) { return(j+k); }
```

In this example (1.1), 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 most 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 positive integers only rather than any possible integer.

Example 2.1 is a relatively simplistic and trivial one, however a more realistic example can be seen in example 2.2:

```
public class GenericArray<T> { (2.2)
    public T[] elems;
    public int next;
    // further details omitted

    /*@ requires t != null &&
           next < elems.length - 1;
       assignable elems[next], next;
       ensures next == \old(next) + 1 &&
              elems[\old(next)] == t &&
              (\forall int i; 0 <= i && i < \old(next);
               \old(elems[i]) == elems[i]);
    */
    public void add(T t) { /* implementation omitted */
        // further details omitted
    }
}
```

This is an example of the generic class `GenericArray` of the generic type `T` with an array used as the underlying data structure that is also of the generic type `T`. The specification present in this example describes the contract for the `add` method of the `GenericArray` class by detailing the pre-conditions (the `requires` clause) and the post-conditions (the `ensures` clause). In this specification, we can see that the `add` method can only be called if the parameter is not null and if the `next` pointer (a pointer determining the next open position in the array `elems`) is at a legitimate position in the array (so as to avoid an `ArrayIndexOutOfBoundsException` exception). In the `assignable` clause, the specification details that both `elems[next]` (the position into which the element to be added will be inserted) and `next` (the next open array index pointer) will be modified by this method. Finally, the `ensures` clause is broken up into three separate sections in order to fully specify the post-conditions that should hold true after the execution of the `add` method. The first part of the `ensures` clause specifies that the `next` pointer should be increased by one with respect to its previous value. The second part of the `ensures` clause specifies that the

first open position in the array `elems` (the previous value of the `next` pointer) should be equal to the object to be inserted, or the parameter used in the call to the `add` method. This is equivalent to saying that the object was actually inserted into the first open position in the array. The final part of the `ensures` clause specifies that every position in the `elems` array up to the position at which the new element is to be inserted (that position being the old value of `next`, `\old(next)`) has maintained its value. This is done by checking that the current value at any position (`elems[i]`) is equal to the old value at this position (`\old(elems[i])`). Thus by formally specifying this `add` method, any developer can look at the specification and understand exactly what the pre and post-conditions for the method are. By understanding the contract of the `add` method, developers should be able to easily create their own implementations of this method.

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

2.3. The `jmlc` Tool

The `jmlc` tool was designed to turn JML specifications into constraint programs. As JML is a formal specification language, these specifications themselves are not executable; however the constraint programs created through the use of the `jmlc` tool can be executed as normal Java byte code utilizing the libraries of the Java Constraint Kit (JACK) (for further information on JACK, see section 3.2). Having an executable formal specification for an application greatly simplifies the development process for software

engineers. With the ability to actually execute the specification, the developer is not only able to confirm the meaning of the specification, but can also validate whether or not any given software module fits its specification. Furthermore, having an executable version of the specification greatly simplifies the process of debugging the specification in order to determine its correctness (Wahls and Krause, 2007; Abdennadher et al., 2002).

2.4 The `jmlc` Tool In Eclipse

In its current version, the `jmlc` tool has been ported into the Eclipse architecture (Eclipse being a very popular and extremely powerful Java based integrated development environment (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, but Professor Tim Wahls is the only one to be working with the `jmlc` tool. Within this Eclipse-based context, when a program is run, the program itself is not actually being run; rather, the `jmlc` tool translates the JML specifications into a constraint program, and these constraint programs are then run utilizing the JACK library (Abdennadher et al., 2002). Instead of using code from a non-executable formal language, this utilization of the `jmlc` tool turns the JML specifications into executable code. The executable code can then be run in its own right and checked for accuracy. The developer's ability to check the meaning of the specification against what was truly meant through this type of "hands on" technique not only makes developing the specification easier, but greatly simplifies the very understanding of the specification.

3. Relevant Literature

Although there is a large amount of literature pertaining to JML, the following is a selection of the most relevant articles to this project.

3.1 jmle: A tool For Executing JML Specifications Via Constraint Programming

The `jmle` tool is a Java based tool created by Professor Tim Wahls and Ben Krause ('08). This tool is an adaptation of the earlier `jmlc` tool (a JML tool to create code that performs runtime assertion checking) in order to compile JML specifications to constraint programs. These constraint programs are then executable via the Java Constraint Kit (JACK) (see section 3.2 for further information about JACK) (Wahls and Krause, 2007). This is a powerful tool, though unfortunately it lacks implementation of many of the new and powerful features introduced into Java with the release of the Java 5 JDK. Amongst these missing features are generics and enumerated types, to list just two. To this end, these will be the features that I will be adding to this tool during the course of my honors project.

3.2. JACK: A Java Constraint Kit

Constraint programming has made substantial progress in recent years. It is now at a point at which many of the essential features are available as libraries, or these features are even embedded in some languages. The need for a constraint library in the Java language is underscored by the ever-growing popularity of the language. To this end, a group of researchers created the Java Constraint Kit (JACK). JACK is made up of three main parts: the constraint handling rules, JCHR (Java Constraint Handling Rules), a tool to aid in the visualization of these rules, VisualCHR, and an abstract search engine to solve the constraint rules, JASE (Java Abstract Search Engine) (Abdennadher et al., 2002).

In any constraint system, the ultimate goal is to remove all the constraints from the constraint store. This constraint system has two stores, one of user-defined constraints and the other of built-in constraints. When a user-defined constraint (from the JCHR file) is activated from the store, it checks for applicability in all of the rules that it appears in. Depending on the type of the firing rule, differing actions ensue. If the firing rule is a simplification rule, all of the matching constraints are consequently removed from the constraint store. In the firing of a propagation rule, all the matching constraints are left in the store. If the rule is a simplification rule, a hybrid of simplification and propagation rules, some constraints are kept in the store while others are removed. If the active constraint has remained in the constraint store after this process has taken place, the system tries another rule in an attempt to remove all the constraints from the store. In the event that all the rules have been tried and the currently active constraint has remained in the store, the system will suspend the constraint until it is reactivated. Upon reactivation, this entire process is repeated in yet another attempt to remove all the constraints from the store (Abdennadher et al., 2002).

The JACK toolkit is an indispensable portion of the `jmlc` tool as the `jmlc` tool relies on this toolkit for the ability to run the generated constraint programs.

3.3 Executing JML Specifications of Java Card Applications: A Case Study

This article is about the use of the `jmlc` tool in order to execute JML specifications in a Java Card application. The goals for this study were to determine the actual efficacy of the `jmlc` tool on a real, production level program, and one that is moderately large and rather complex at that (Cantano and Wahls, 2008). Overall, the project was a success, leading to changes in both the implementation of the `jmlc` tool and in the specifications

found in the Java Card electronic purse application. Some were problems encountered, including the significant amount of time needed to diagnose problems. This was partly due to the lack of information in error messages from the `jmle` tool, wherein the only message given on failure was that the failure came in evaluating either the `requires`, `modifies`, or `ensures` clauses.

This article is useful in determining what changes can be made to the `jmle` tool to make it a more powerful and fully useable tool, especially as it relates to actual use in industry and with larger and more complex applications.

4. Progress Thus Far

Thus far, I have divided my efforts in adding generic capabilities to the `jmle` tool integrated into the Eclipse IDE into three main areas. The first area that I have been working with is in the modification of a class, created by Professor Tim Wahls, used internally by Eclipse. This class is used by a modified version of the Eclipse compiler to create a constraint program, which can then be run via the JACK constraint kit. The second area that I have been working with is in the modification of the supporting code for the `jmle` tool. The third area is in the implementation of immutable generic JML model classes, which are heavily used by the `jmle` tool.

Even though the progress has been in three main areas, this is not to say that these sections are completely distinct. They overlap greatly, especially progress with the JML model types and the supporting code for the `jmle` tool. Therefore, progress in one area often directly correlates with progress in another area.

4.1. Progress In Eclipse

My work with the Eclipse IDE has focused on one class: the `ExecVisitor` class. This class, as mentioned previously, was developed by Professor Tim Wahls in order to bring the functionality of the `jmlc` tool into the Eclipse environment. The `ExecVisitor` class is utilized by a modified version of the internal Eclipse compiler, and this compiler takes JML specified code and compiles it to a constraint program using the `jmlc` tool.

As the `ExecVisitor` class was originally written without consideration for generics, there were a few issues that arose with attempting to use this code with generic classes and methods. Initially, the created constraint code classes had no generic type parameters. To this end, I modified the `ExecVisitor` class to add type parameters (either one or many) to the created class.

After adding the ability to create constraint classes with generic type parameters, there were still numerous syntax errors arising with the created constraint code. Again, this was due to the fact that the `ExecVisitor` class was originally created without thought for the implementation of generics. There were numerous methods in the created constraint code that were using the full name of the class, including the generic type parameter(s), as a prefix for the full method name. A case in point is seen in example 4.1.

```
public void GenericClass1<T>$spec$settheList(...) {} (4.1)
```

Clearly, this is an illegal syntax within the Java language, and something that needed to be fixed. In the situations where these generic type parameters were being inserted into the method names, I found the locations in the `ExecVisitor` class where these method names were being created and then was able to easily strip out the generic type parameters, eliminating the syntax errors from the created constraint code.

4.2. Progress With Supporting Code To The `jml` Tool

The `jml` tool is the central focus of this project, as it is this tool that I am trying to make compliant with generics. Thus far, my efforts have revolved around making some of the other classes used in this tool generic. These classes were originally written without generic support, so the implementation of generic types into these classes is a critical step in the overall progress of this project. One example of such a conversion that I have implemented is in the `BagArrayList<T>` class. In the generification of this class, I also had to generify the classes used to hold elements in the `BagArrayList` class, those being the `EqualsBagElem` and `ObjectBagElem` classes, as well as the implemented interface to these two classes, `BagElem`. Although this might seem like a straightforward task, there were some design decisions to be made with regards to some of the methods.

My further work with the `jml` tool has been focused on the `JMLTool` class, a class that defines a number of static methods. These methods are heavily used in the JCHR constraint-handling rules (one of the essential parts of JACK). As the constraint-handling rules and their associated code (the static methods of the `JMLTool` class) lie at the center of the `jml` tool, it is in this area that the majority of the remaining work lies. Thus far, I have been making progress in the generification of these static methods and the correlating constraints. My progress has revolved mostly around changing the methods to use generic types instead of non-generic types. This has involved fixes such as changing instances of `JMLCollection` into `JMLCollection<T>`, adding the parameterized type (these changes are far from done, although they are well underway), and the generification of the `BagArrayList` class (mentioned previously), just to name two of the steps that I have

taken towards the implementation of generics. At the present time, I have made very few changes to the constraint-handling code its self, but this is the next step that I will be taking in order to match the all of the changes that I am currently implementing within the `JMLTool` class.

4.3 Progress With Generic JML Model Classes

One of the larger hurdles that I face in implementing generics into the `jmlc` tool is the heavy dependence on the immutable JML model classes. These classes were all originally written without any generic support, so I knew that I would have to be re-implementing them in a generic manner. Luckily, before I began working on the generification of these model classes, David Cok, one of the more active participants in the JML4 project, began the implementation of these generic classes. Although his work thus far has only involved interfaces and abstract classes, I have been able to use this work as a starting point in the implementation and testing (utilizing JUnit test classes) of two generic JML model classes, `JMLObjectSet<E>` and `JMLObjectListNode<E>`. Although these two classes make up only a small portion of the generic JML model classes, having implemented and tested these for correctness will greatly aid in the creation of the remaining model type classes. In fact, as there is a large amount of repetition amongst some of these classes, these two proven classes can be used as templates in the creation of some of the generic JML model type classes, especially in the utilization of a script to create some of these classes.

Although these two classes have been tested, some details remain that are not completely finalized. For example, the design of immutable iterators has not yet been

finalized. Although the current design seems to work with the two concrete classes that I have implemented and tested (those being the aforementioned `JMLObjectSet` and `JMLObjectListNode`), the design still needs to go through much scrutiny in the future in order to be certain that the design decisions made are the correct ones, especially with regards to future compatibility issues.

5. Remaining Work and Challenges Ahead

Although much work has been done up to this point, there is still a large amount of work left in order to complete the implementation of generics into the `jmlc` tool within the Eclipse environment. Much more effort needs to be focused on the constraint-handling rules and the correlating code within the `JMLTool` class, amongst others. As well, in order to do this I will need to complete the generification of the JML model type classes. This will likely be partially accomplished through some sort of standardized script, which will be based off of the two JML model type classes I have implemented and tested.

Along with this amount of work comes numerous challenges. One of the ever-present challenges I face is the sheer complexity of this project. As I continue to work with this project, I become more and more knowledgeable with the ways in which it works; even so, there are still details that I am learning about this project through the help of my project advisor, Professor Tim Wahls. As well, there are some design decisions in this implementation that need to be determined. One such decision, mentioned earlier, is if the current design of immutable generic iterators is correct or whether it should be changed in some way. There will most likely be more design decisions along the way as I continue to generify the JML model type classes.

Even in light of these issues, I firmly believe this project's goal to be achievable in due time. In fact, if the goal should be achieved earlier than expected, I would like to even extend the bounds on this project to include the addition of enumeration type functionality to the `jmle` tool, bringing it even closer to full Java 5 JDK compatibility. This goal will, however, have to wait and see, dependant on how the main goals of this project run their course.

Given that Professor Tim Wahls has had extensive experience with the JML4 language and that he was the one to implement the `jmle` tool into an Eclipse based environment, I will continue working on this project under his guidance. Additionally, I have enlisted the support of Professors Grant Braught and John MacCormick as readers for this project.

6. References

1. 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.
2. 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.
3. N. Catano and T. Wahls. Executing JML Specifications of Java Card Applications: A Case Study. Accepted for the 24th ACM Symposium on Applied Computing, Waikiki Beach, Honolulu, Hawaii. March 8 - 12, 2009.
4. B. Krause and T. Wahls. `jmle`: 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 2007.
5. 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.
 6. G. T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. (2008). JML Reference Manual. Retrieved December 2, 2008, from http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html.