

# 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.

March 20, 2009.

The Department of Mathematics and Computer Science at Dickinson College hereby accepts this senior honors thesis by Adrian Kostrubiak, and awards departmental honors in Computer Science.

---

Professor Tim Wahls (Advisor)

---

Date

---

Professor Grant Braught (Committee Member)

---

Date

---

Professor John MacCormick (Committee Member)

---

Date

---

Professor David Richeson (Department Chair)

---

Date

Department of Mathematics and Computer Science  
Dickinson College

March 2009

## Abstract

# Integration of Java Generics Into The `jml` Tool Within The Eclipse IDE

by  
Adrian Kostrubiak

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. JML is used to formally specify the behavior of modules in a Java program. Due to their nature, formal specifications cannot actually be executed. However, the `jml` tool created by Professor Tim Wahls has allowed the execution of these formal specifications through converting them into constraint programs. Unfortunately, previous to this work, neither JML nor any of the supporting tools were compatible with many of the newer Java 5 JDK features, such as generics and enum types. The goal of this research is to bring the functionality of generics into the `jml` tool. Although there were numerous unexpected complications, limited generic functionality has been brought to the `jml` tool. The aforementioned success has shown that the full integration of generics into the `jml` tool should not be much more work at this point. However, for the `jml` tool to acquire complete Java 5 JDK compatibility, much more work needs to be done.

## **Acknowledgements**

Firstly, I would like to thank Professor Tim Wahls, my advisor on this project. Without his help and the many hours he has already spent working with JML and the `jml` tool, my project could never have been possible. Secondly, I would like to thank Professors Grant Braught and John MacCormick, both of whom were gracious enough to sign up as readers for this thesis. Lastly, I would like to thank my family and all of my friends who have supported me while I have spent many hours working on this massive project.

## Table Of Contents

Title Page	i
Signature Page	ii
Abstract	iii
Acknowledgements	iv
Table Of Contents	v
<b>Chapter 1: Introduction</b>	<b>1</b>
1.1. Introduction	1
<b>Chapter 2: Background Information</b>	<b>3</b>
2.1. Java Generics	3
2.2. Java Modeling Language (JML)	3
2.3. The <code>jmlc</code> Tool	6
2.4. The <code>jmlc</code> Tool In Eclipse	6
<b>Chapter 3: Relevant Literature</b>	<b>8</b>
3.1. <code>jmlc</code> : A tool For Executing JML Specifications Via Constraint Programming	8
3.2. JACK: A Java Constraint Kit	8
3.3. Executing JML Specifications of Java Card Applications: A Case Study	9
<b>Chapter 4: Results</b>	<b>11</b>
4.1. Eclipse Internal Classes	11
4.2. JML Model Classes	12
4.3. Design Decisions and Implementations	13
4.3.1. Genericizing Static Methods	13
4.3.2. Immutable Iterators	14
4.3.3. Temporary Solutions For Integration and Testing	16
4.4. Other Supporting Classes	16
<b>Chapter 5: Conclusions</b>	<b>18</b>
<b>Chapter 6: References</b>	<b>19</b>

# Chapter 1

## Introduction

### **1.1 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 correctness of the software modules that they have written. One of the directions that this search has taken is in the area of formal specifications, which are 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 tools for runtime assertion checking and general code documentation (Leavens et al., 2000), amongst many others. 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, the goal of this research has been focused on the integration of Java generics into the `jmlc` tool, a tool that is used for creating executable versions of the specifications. This paper aims to document the process of adding generic functionality into the `jmlc` tool.

An overview of the relevant technologies will be presented first, followed by a brief overview of some of the more relevant literature. After this, the cursory results will be

presented, along with some information about the design decisions. Finally, conclusions and the consequences of the results will be presented.

## Chapter 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 Java Modeling Language**

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 &&
           (\forallall 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 (Chalin et al., 2007), 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

jmle 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 as well greatly simplifies the very understanding of the specification.

## Chapter 3

### Relevant Literature

#### **3.1 jmlc: A tool For Executing JML Specifications Via Constraint Programming**

The `jmlc` 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.

## Chapter 4

### Results

#### **4.1 Eclipse Internal Classes**

One of the distinctive areas in which there has been much progress towards the goal of adding generic types to the `jmlc` tool has been in the modification of one internal Eclipse class, the `ExecVisitor` class. The `jmlc` tool uses this internal class in order to translate the JML annotations into constraint code. Like much of this project, the code for this internal class was originally written without consideration for the use of generic type parameters. Thusly, upon initial testing of a class with generic type parameters in `jmlc`, the tool was creating constraint code with numerous syntactical errors. Amongst the issues occurring with the generated constraint code was that the class that was created as constraint code was not parameterized. To this end, I modified the `ExecVisitor` class to add type parameters (either one or many) to the created class.

Once the created class had been parameterized, numerous syntax issues still existed with this created constraint code. 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 various locations in the `ExecVisitor` class where these

method names were being created and then was able to easily strip out the generic type parameters with a helper method, eliminating the syntax errors from the created constraint code.

The last area in which I have worked on making this Eclipse internal class compatible with generic types has involved sections of code that previously had relied on certain variables being of a specific type. Again, as this internal class was written without consideration of generics, when these variables changed their type to represent the fact that they were now representing generic classes, this caused the internal class to not compile certain sections of JML as constraint code, as it should have been doing. However, after tracing these few cases down, the internal class was once again compiling every it should as constraint code.

## **4.2 JML Model Classes**

One of the key components to the JML4 project is JML model classes. This is a collection of classes that are used to represent the type of traditional mathematical notation that one would have found in previous behavioral interface specification languages not associated with Java. One of the goals of JML is to have a very Java-like syntax, both making the learning curve an easy one and as well allowing Java programmers to easily pick up this formal language due to the familiarity of the syntax. These model classes are the Java implementations of the traditional mathematical notations, allowing JML to continue to use a syntax very similar to Java's own syntax.

One of the biggest requirements in adding generic functionality to the `jmlc` tool is the creation and testing of generic versions of these model classes. Two of the most

frequently used of these model classes, `JMLObjectSet` and `JMLObjectSequence`, have both successfully been genericized, along with the necessary node class, `JMLListObjectNode`, as well as all the directly related interfaces, super classes, and so forth. After successfully creating the generic versions of these two model classes, both were thoroughly tested and adjusted to be sure that the implementations were correct using the `jml` tool within the Eclipse environment. This process continued until both of these classes were able to successfully pass a rigorous test suite.

The creation of these two generic model classes and testing of them using the `jml` tool in Eclipse has shown that creating a template for the further creation of the rest of the model classes should not prove to be terribly difficult. As well, having successfully debugged these two classes in the `jml` tool now means that when the rest of the generic model classes are created, they ought to integrate with the current system seamlessly.

### **4.3 Design Decisions And Implementations**

Through the course of the project, there have been a few important design decisions and implementations. The following details three of the larger and more important design decisions that have been made during the course of integrating generics into the `jml` tool.

#### *4.3.1 Genericizing Static Methods*

In working with genericizing the JML model classes, there are a number of static methods involved in these classes, all of which needed to be genericized. However, due to the fact that non-static variables cannot be accessed from static contexts, and vice-versa, simply declaring these methods as generic did not suffice. A good example can be found in

the JML model class `JMLObjectSequence<E>`. In this class there is a static `singleton()` method, which is used to create a new `JMLObjectSequence` with the supplied element (the single parameter). Although logically it would seem reasonable to use the parameterized type of the class, `E` (as seen in example 4.2), as the formal parameter for the method call, this unfortunately is not allowed by Java syntax due to the fact that non-static classes (and variables) cannot be referenced in static contexts (which this is), and vice versa.

```
public class JMLObjectSequence<E> {                               (4.2)
    // further details and implementation omitted
    public static JMLObjectSequence<E> singleton(E obj) {}
    /* This fails, as non-static E is being referenced in a
       static context, which is not allowed in Java */
}
```

Some research into the finer details of the implementation of generics in Java yielded interesting results: just as a class or variable can be generic, so too can a method be generic. One can declare a method with a formal type parameter. A reimplementaion of example 4.2 with the `singleton()` method being declare generic can be seen in example 4.3:

```
public class JMLObjectSequence<E> {                               (4.3)
    // further details and implementation omitted
    public static <F> JMLObjectSequence<F> singleton(F obj) {}
    /* This compiles fine and produces the expected results */
}
```

#### 4.3.2 *Immutable Iterators*

Preliminary work by other researchers associated with the JML4 project had the outlined generic model classes created in such a way that all objects of these classes were to be immutable (Cok, 2008). This is to say that once an object has been instantiated, it cannot be changed in any way. Such a design decision has many implications. One example of the implications that an immutable object design can have on a class can be seen easily if we are

to examine any of Java's collection classes (these classes are certainly not immutable, however they serve as a good example for the implications of immutability). In the normal Java implementation of `LinkedList` (that being the non-immutable implementation), the `add()` method returns either a `boolean` (dependent on success of the `add()` call), or simply `void`, depending on the parameter(s) used to this method call. However, if we wanted to create an immutable version of this `add()` method, this clearly poses certain problems, as adding an object to a `LinkedList` is by definition changing the original object. Thusly, in an immutable version of the `LinkedList`, the `add()` method would have to produce no side effects. This is accomplished by having the `add` method instantiate a new copy of the `LinkedList` containing all the elements from the previous `LinkedList` and of course the new element to be added. This in turn means that the `add()` method must have a return type of `LinkedList`, as opposed to either `boolean` or `void`. The impetus for such a design decision lies in the fact that only immutable methods can be used in formal specifications.

All the model classes have iterators in order to be able to iterate through the elements of the collection in a straightforward manner. In converting the model classes into generic model classes, the question of immutable iterators was raised: should iterators be immutable, and if so, how exactly would this work? After considering this question for quite a while, it was decided that immutable iterators were not only unnecessary, but as well simply an unreasonable as a concept. As iterators inherently have side effects, they cannot possibly be immutable. Amongst the reasons for which it was decided that iterators need not be immutable is included the fact that should one ever want to use an iterator in a specification, it would be just as easy, if not even easier to simply use the JML universal quantifier,

\forall. This effectively covers the use of the iterator within specifications, and makes the use of an iterator in a specification completely unnecessary.

### *4.3.3 Temporary Solutions For Integration and Testing*

For the temporary ease of integration and testing within the current system as it is set up, the new generic model classes, which are supposed to be immutable, have had their immutable aspects removed. As well, they have been put into the same package as the normal model classes, just in place of the non-generic counterparts to these new generic classes. Only once the entirety of the JML model classes has been converted into generic forms, the new generic models should be placed in a separate and unique package (in order to distinguish these generic and immutable models from the non-generic and mutable versions), and the immutability of these classes regained.

Although this is just a temporary measure, it has been one of the keys to being able to see significant amounts of progress without having completed a tremendous amount of work, much of which is somewhat extraneous. This has been highly beneficial from a development standpoint, for now it is possible to judge and actually see how much effect smaller changes have, whereas previously to see any amount of quantifiable change in the project would have required, at a minimum, very large amounts of work.

## **4.4 Other Supporting Classes**

Much other work has also been done with various other supporting classes, most directly relating with the `jmlc` tool. Within this tool, there is a heavy amount of reflection being used. One of the ways in which reflection is used is by the creation of classes using the

class `Class`. One method of creating a new class object is done via the utilization of the `forName()` static method, which takes a string parameter specifying the name of the class to be created. Unfortunately, with this method, there is no way to create a generic version of a class that is to be instantiated. Making this a problem, many classes that use this method call within the `jmle` tool were created without consideration for generics. These two factors together resulted in a situation where there were numerous method calls similar to `Class.forName("java.util.LinkedList<E>")`. Given the angled brackets denoting the parameterized type, this clearly is not a legitimate syntax, as a class cannot have angled brackets in its name. Therefore, there were numerous instances throughout the supporting `jmle` code where the parameterized type of a class needed to be stripped out from the parameter to the call to `Class.forName()`. This ultimately proved greatly more complex than originally suspected as calls to `Class.forName()` are littered throughout the `jmle` supporting code in a variety of classes. Tracking down all of these calls proved to be a truly time consuming process.

## Chapter 5

### Conclusions

#### **5.1 Current Progress and Moving Forward**

All in all, after many a stray path, this project has been successful thus far in the implementation of generics into the `jm1e` tool within the Eclipse IDE. Even though many of the model classes remain to be genericized, the work completed thus far has proven that at this point it is simply enough to genericize the model classes and they will work with the tool in its current implementation.

Of course, bringing the `jm1e` tool and at the same time the entire JML4 project up to Java 5 JDK compatibility will still require much work, as there are numerous features that were added to Java 5 which have yet to be implemented in JML. Amongst these features is included `enum` types and autoboxing (automatically creating an Integer object around a primitive `int` when necessary), to name just two.

This project has laid a solid groundwork for the hopefully soon coming complete Java 5 compliance for both the JML4 project and the `jm1e` tool. Only once this migration has occurred can the `jm1e` tool, along with all the other tools associated with the JML4 project make the move away from mere academic pursuits to being very powerful and robust suites that developers would be able to leverage in the implementation of the design by contract ideals in their programs.

## **Chapter 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. P. Chalin, P. R. James, and G. Karabotsos. An Integrated Verification Environment for JML: Architecture and Early Results. *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 47-53, September 2007.
5. D. R. Cok. Adapting JML to generic types and Java 1.6. *Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008)*, pages 27-34, November 2008.
6. 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.
7. 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.
8. 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](http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html).