# The Java Modeling Language

A study and an implementation of JML

W I L H E L M   K Ä R D E

KTH Computer Science
and Communication

Bachelor of Science Thesis
Stockholm, Sweden 2011

# The Java Modeling Language

A study and an implementation of JML

W I L H E L M   K Ä R D E

Bachelor’s Thesis in Computer Science (15 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2011
Supervisor at CSC was Mads Dam
Examiner was Mads Dam

URL: www.csc.kth.se/utbildning/kandidatexjobb/datateknik/2011/
karde_wilhelm_K11056.pdf

Royal Institute of Technology
*School of Computer Science and Communication*

**KTH** CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

## Abstract

This is a study on the Java Modeling language, presenting its main features and applying them onto an algorithm based upon Binary Decision Diagrams. JML is a Design by Contract tool for java. The principal idea behind Design by Contract is that clients calling methods in a class have a contract with each other. These contracts consist of pre- and post-conditions that are to be validated before and after the execution of any method with such a contract.

As interesting as it was studying JML, at the implementation stage it became clear that the JML2 tool set was not upgraded for any java upgrades beyond java 1.4. Using generic types and other language features such as the for-each loop was not recognized by the JML compiler.

However, once up and running with a J2SE4 environment, many advantages could be discovered. The runtime assertion checker worked great as a bug prevention tool and the JML specifications serve as a good way to document code properly. More important though is the way it forces the developer to take into consideration all the different relationships between classes and their methods and also to define set invariants to hold for these.

## Sammanfattning

Det här är en studie gjord på “The Java Modeling Language” genom att presentera dess viktigaste funktioner samt applicera dessa på en algoritm baserad på ”Binary Decision Diagrams”. JML är ett ”Design By Contract” verktyg för java. Idén bakom DBC är att en klient och en metod ur en klass har ett kontrakt med varandra. Detta kontrakt realiseras genom pre- och post-vilkor som måste uppfyllas före och efter exekveringen av metoden.

Den intressanta studien fick en trist uppföljning när det visade sig att JML kompilatorn inte är kompatibel med java 1.5 och senare. Detta resulterade i kompilatorn inte kände igen generiska typer och ”for-each” loopar t.ex.

Däremot, om man är villig att koda i java 1.4 så kunde flertalet fördelar med JML upptäckas. JMLs ”runtime assertion checker” fungerar utmärkt för att förhindra och upptäcka buggar. Specifikationerna fungerade dessutom som bra kod dokumentation. Men viktigast av allt var hur JML tvingar en att tänka på relationen mellan klasser och metoder och invarianter som ska hålla för dessa.

# Contents

1 Introduction…… 1

2 Background…… 2

2.1 The Java Modeling Language…… 2

2.2 Binary Decision Diagrams…… 4

2.3 ROBDDs……5

3 Implementation……6

3.1 The ROBDD Algorithm……6

3.2 JML Implementation……8

4 Results……9

4.1 No Support for Generic Types……9

4.2 Implementation Difficulties……10

4.3 Implementation Analysis……11

5 Discussion / Conclusions……14

Bibliography……15

# 1. Introduction

This essay focuses on JML “Java Modeling Language”. JML is a DBC “Design By Contract” tool for Java. The principal idea behind DBC is that a class and its clients have a “contract” with each other. The client must guarantee certain conditions before calling a method defined by the class, and in return the class guarantees certain properties that will hold after the call.[1]  JML is targeted at providing  a comprehensive specification of both interfaces (syntax) and behavior (semantics) for every aspect of Java and, at the same time, to retain an easy-to-read format. [4]

Design by contract is a fairly new way of developing software. It breaks away from the standard way of programming which makes it interesting and will render a new perspective on developing software. The principal goal of this essay is to cover as much as possible about JML and apply what learnt on an algorithm, so as to get a good feel for its practicality. This will be done on a binary decision diagram algorithm.  Binary decision diagrams (BDDs) are another way of representing Boolean Functions. [3]

# 2. Background

Before implementing JML into actual code, there will be some background information about JML and Binary Decision Diagrams (BDDs). This is to be considered only as an introduction to both areas as this background information only scrapes the surfaces of both subjects.

## 2.1 The Java Modeling Language

As mentioned in the introduction, JML is to be considered a Design by Contract tool for the java language. Benefits when working with JML are among other factors: More precise description of what the code does, Efficient discovery and correction of bugs, Reduced chance of introducing bugs as the application evolves, Early discovery of incorrect client usage of classes, Precise documentation that is always synced with application code.[8]

JML specifications are written in special annotation comments, which start with an at-sign (@). [1] Examples are illustrated in figure 1. Thus when running a program in a standard java compiler, the JML specifications are treated as any other comments. To verify that the specifications are correct a JML compiler and a JML runtime assertion checker (RAC) are needed. The JML compiler (jmlc) behaves like Java compilers, such as javac, by producing Java bytecode from source code file. The difference is that it adds assertion checking code to the bytecodes it produces. Only the classes that are to be runtime assertion checked need to be compiled with jmlc. [1]

The contract between a client and a class requires pre- and post-conditions. In the example code that follows, the precondition and postcondition are specified by **requires** and **ensures** respectively. The method specification begins in line 8 with a Java-style privacy modifier and the keyword **normal_behavior**. The latter requires the method to terminate normally, i.e. without exceptions. [4]

```
public class Book {
private /*@ spec_public @*/ boolean lent ;
}
public class Library {
private Collection coll ;
/*@ public normal_behavior
@ requires coll . contains (b)
@ && !b. lent ;
@ ensures b. lent ;
@*/
public void lend (Book b);
}
```

Figure 1: An example of JML syntax along with Java code.

In figure 1, two JML specifications are displayed. The first one, **/*@ spec_public @*/**, declares the following Boolean variable *lent* to be used as a public specification variable. This means that *Lent* can now be used in any other specification, as shown in the second specification. As it is declared as a private variable, it would not be possible to use it in other specifications otherwise. The second specification is a method specification for the method *lend(Book b)*. The precondition states that the requirements to call this method is that **coll.contains(b) && !b.lent** must hold. Likewise, the postcondition states that if the method terminates then **b.lent** must be true.

Another cornerstone in the JML specification language is the use of invariants. An invariant is a property that should hold in all client-visible states. It must be true when control is not inside the object's methods. That is, an invariant must hold at the end of each constructor's execution, and at the beginning and end of all methods.[1]

A few important features that are essential to know about JML are the use of quantifiers, important key words such as **\result** and **\old()**, implication arrows and the key word **pure** which declares a method pure. Only pure methods can be used in specifications. A method can only be declared pure if it offers no side-effects. The **\result** key word represents the return value from a method and **\old()**, which takes a parameter, represents the value of that parameter from a previous state. Examples of an implication arrow that can be used in specifications is, a ==> b (a implies b), and there are a few others which are shown in table 2.

| Syntax | Meaning |
|---|---|
| a ==> b | a implies b |
| a <== b | a follows from b (b implies a) |
| a <==> b | a if and only if b |
| a <=!=> b | not (a if and only if b) |

Table 1. Some implication arrows that can be used in JML specifications.

Finally, JML supports several kinds of quantifiers in assertions: a universal quantifier **(\forall)**, an existential quantifier **(\exists)**, generalized quantifiers **(\sum**, **\product**, **\min**, and **\max)**, and a numeric quantifier **(\num of)**.[1]

## 2.2 Binary Decision Diagrams

Binary decision diagrams (BDDs) are another way of representing Boolean Functions. Boolean Functions are an important descriptive formalism for many hardware and software systems, such as synchronous and asynchronous circuits, reactive systems and finite-state programs. Representing those systems in a computer in order to reason about them requires an efficient representation for Boolean Functions. [3]

To obtain a decision diagram, we perform Shannon expansions ( $T = x \rightarrow [x_1/1], [x_0/0]$ ) on one variable at a time in a Boolean function. The expression $T = x \rightarrow [x_1/1], [x_0/0]$ means that a variable substitution is performed on variable x in function T. In the case of Shannon expansions the substituted variables holds a value denoted by the number below the slash. In this expression x is replaced by the variables $x_1$ and $x_0$ who are assigned to holding the values 1 and 0 respectively.

In the example shown in figure 2, $X_1$ is the first variable in the function $T = (X_1 \Leftrightarrow Y_1) \Lambda (X_2 \Leftrightarrow Y_2)$ and it is the replaced by $T_1$ and $T_0$, assigned to holding the values 1 and 0 respectively. This is the same as performing $T = X_1 \rightarrow [T_1/1], [T_0/0]$.

Evaluating the second variable $Y_1$, the substitution performed is equivalent to $T = Y_1 \rightarrow [Tx_1/1], [Tx_0/0]$. In this substitution the variable x denotes which substitutions that already have been made on previous variables in the function T. Therefore, i.e. $T_{01}$ is equivalent to the first variable ($X_1$) in function T being replaced by the value 0 and the second variable ($Y_1$) by the value 1. Furthermore, $T_{0001}$ denotes the first three variables in T being replaced by 0 and the last one by the value 1.

If an absolute value for the Boolean function T can be determined before expansions have been made on all variables, no further expansions are necessary for that particular branch. $T_{01}$ evaluates to 0 for the Boolean function T regardless of what values the last two values take on. Further expansions on $T_{01x}$ is therefore not necessary as they all evaluate to the same value for the function T, namely 0.

Evaluating the function $T = (X_1 \Leftrightarrow Y_1) \land (X_2 \Leftrightarrow Y_2)$ yields the expressions in figure 2 and further down in figure 3, the expressions are shown as a tree. Such a tree is called a decision tree. [3]

- $T = (X_1 \Leftrightarrow Y_1) \land (X_2 \Leftrightarrow Y_2)$
- $T = X_1 \Rightarrow T_1, T_0$
- $T_0 = Y_1 \Rightarrow 0, T_{00}$
- $T_1 = Y_1 \Rightarrow T_{11}, 0$
- $T_{00} = X_2 \Rightarrow T_{001}, T_{000}$
- $T_{11} = X_2 \Rightarrow T_{111}, T_{110}$
- $T_{000} = Y_2 \Rightarrow 0, 1$
- $T_{001} = Y_2 \Rightarrow 1, 0$
- $T_{110} = Y_2 \Rightarrow 0, 1$
- $T_{111} = Y_2 \Rightarrow 1, 0$

| $\land$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| $\Leftrightarrow$ | 0 | 1 |
|---|---|---|
| 0 | 1 | 0 |
| 1 | 0 | 1 |

Figure 2: Shannon expansions on the Boolean Function $T = (X_1 \Leftrightarrow Y_1) \land (X_2 \Leftrightarrow Y_2)$ along with its truth tables for $\Leftrightarrow$and $\land$.

A Binary Decision Diagram is a rooted, directed acyclic graph with

- one or two terminal nodes of out-degree zero labeled 0 or 1, and
- a set of variable nodes of out-degree two. The two outgoing edges are given by two functions *low(u)* and *high(u).* (In figure 3 these are shown as red and black lines respectively). A variable *var(u)* is associated with each variable node.[3]

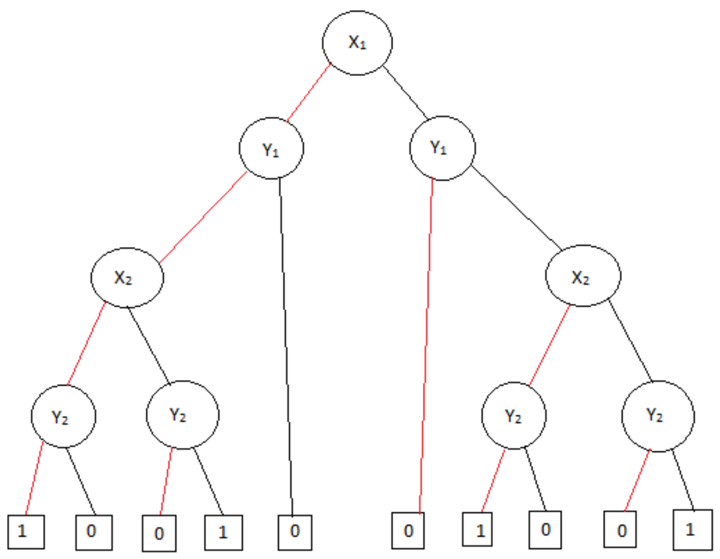

Figure 3: A decision tree for the Boolean Function T = $(X_1 \Leftrightarrow Y_1) \wedge (X_2 \Leftrightarrow Y_2)$

## 2.3 Reduced Ordered Binary Decision Diagram

Reduced Ordered Binary Decision diagrams (ROBDDs) have some interesting properties. They provide compact representations of Boolean expressions, and there are efficient algorithms for performing all kinds of logical operations on ROBDDs. They are all based on the crucial fact that for any function **f** : $\boldsymbol{B}^n \rightarrow \boldsymbol{B}$ there is exactly one ROBDD representing it. This means, in particular, that there is exactly one ROBDD for the constant true (and constant false) function on $\boldsymbol{B}^n$ : the terminal node 1 (and 0 in case of false). Hence, it is possible to test in constant time whether an ROBDD is constantly true or false. [3]

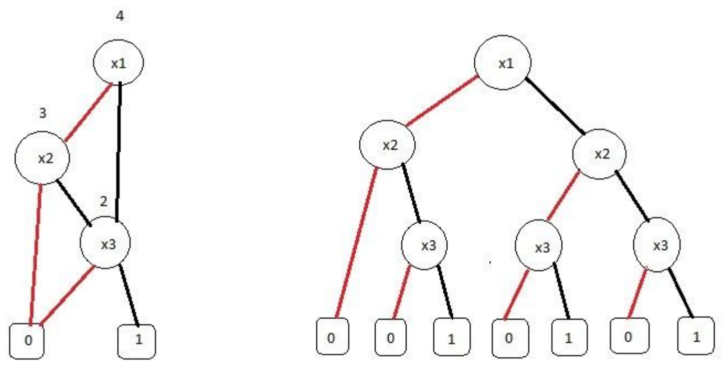

Figure 7: To the left is a ROBDD for the Boolean expression ((x1 || x2) && x3). As a reference the BDD for the same Boolean expression is shown to the right. In the ROBDD the index of each variable is shown above their variable name. Indexes of the terminal nodes 0 and 1 are 0 and 1 respectively.

A BDD is Ordered (OBDD) if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < \ldots < x_n$. An (O)BDD is Reduced (R(O)BDD) if

- (**uniqueness)** No two distinct nodes *u* and *v* have the same variable name and low- and high-successor, i.e.,

  *var(u) = var(v), low(u) = low(v), high(u) = high(v),* implies *u = v*
  and
- (**non-redundant-test**) No variable node *u* has identical low- and high-successors, i.e.,

  *low(u) != high(u).* [3]

## 3. Implementation

In this section, the foundation for the implementation is mapped out. First the algorithms for this work are introduced followed by the introduction of relevant JML considerations for the algorithms.

## 3.1 The ROBDD Algorithm

The algorithm is split into two parts. Part one of the algorithm is shown in figure 4 below. This recursive function *Build* is the core of the algorithm and it takes a Boolean expression as a parameter on which to perform the Shannon expansions as previously mentioned. The second part is an algorithm whose main purpose is to make sure that the ROBDD being constructed is reduced. Hence, its objective is to make sure that the ROBDD being constructed follows the uniqueness as well as the non-redundant-test throughout the construction. Only through the second algorithm is a new Node created and inserted into an array from which the final ROBDD can be read from as seen in figure 5.

```
Build(t, i)
1: **if** i > n **then**
2:         **if** t is false **then return** 0 **else return** 1
3: **else** $v_0$ ← Build (t[$x_i$/0], i +1)
4:       $v_1$ ← Build (t[$x_i$/1], i +1)
5:       **return** GetIndex(i, $v_0$, $v_1$)
6: **end** Build
```

Figure 4: The function Build(t, i)

The parameter t represents the Boolean expression from which we build the tree. An input example of *t* would be "x1&&x2||x3". The parameter *i* represent the index of the current variable on which a Shannon expansion is performed in lines 3 and 4. In line 1 this index is compared with the variable *n*, representing the number of variables in *t*, so when *i* is greater than *n* then all variables have been given a value trough Shannon expansions and *t* can be evaluated as true or false. As previously mentioned, Shannon expansions are performed in line 3 and 4. This is done recursively so that the Shannon expansions are done on one variable at a time in the correct order. The first time a value is assigned to $v_0$ and $v_1$ is when all variables in the expression *t* have been assigned a value (i.e. t = 0&&0||0 instead of t = x1&&x2||x3). The algorithm then calls GetIndex which creates a new Node with *var* = i, *low* = $v_0$ and *high* = $v_1$ if and only if the uniqueness as well as the non-redundant-test requirements are met. The variable *var* is the variable number of the Node. *Low* and *high* are the indexes to the respective low- and high-branch Node in an array. The first two indexes 0 and 1 are reserved for the terminal nodes 0 and 1 respectively. The first Node will therefore have index 3 in the array with low- and high-values of 0 and 1 depending on what they evaluate to. Through GetIndex new Nodes are created if necessary and saved in an array. It is from this array that the final ROBDD can be read. An example of such a representation of a tree in a table is shown in figure 5.

| Node(index) | var | low | high |
|---|---|---|---|
| 0 | 4 | | |
| 1 | 4 | | |
| 2 | 3 | 0 | 1 |
| 3 | 2 | 0 | 2 |
| 4 | 1 | 3 | 2 |

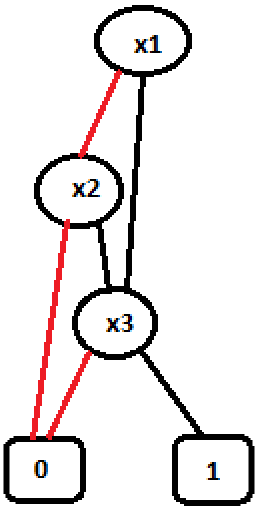

Figure 5: To the left an ROBDD as saved in an array with Nodes : Node → (var, low, high) from the Boolean expression ((x1 || x2) && x3). Index 0 and 1 are represented by the terminal nodes 0 and 1 respectively. Each node is assigned an index variable *var* which is the number of variables in ordering plus one.

The second part of the algorithm is as mentioned the GetIndex method shown in figure 6. In line 1 the algorithm tests that the non-redundant requirement is met. Thereafter, in line 2, the uniqueness requirement is checked so that finally in line 4 a new node can be created and inserted in both table T and H if all requirements hold. Table H is only used to quickly lookup if a Node already exists as shown in line 2 and 3. Note that in line 3 and 6, the return values are the indexes that are to be passed on as values for $v_0$ and $v_1$ in *Build.* In line 1 no new Node is created as the *low* and *high* value are not permitted to be the same, hence their value is simply returned to Build.

```
GetIndex(i, l, h)
1: if l = h then return l
2: else if member(H, i, l, h) then
3:         return indexOf(lookup(H, i, l, h))
4: else u ← add(T, i, l, h)
5:         insert(H, i, l, h)
6:         return indexOf(u)
```

Figure 6: The function GetIndex(i, l, h)

T and H represent two tables saved as ArrayList<Node> and HashMap<String, Node> respectively. Table T contains information about all nodes in the ROBDD and table H is used to lookup existing nodes in the ROBDD quickly. The String key in table H is the combined lettering of the values from the respective Node that it is key to. So a Node with *var* = 3, *low* = 0 and *high* = 1 will have the String key “301”. A Node holds the three values *var, low* and *high.* An example of table T is shown in figure 5.

## 3.2 JML Implementation

From the algorithm definition there are already a few JML implementations that need be taken into consideration. Perhaps the most important one to consider is regarding the uniqueness and non-redundant-test requirements. Since these requirements should hold throughout the entire lifetime of the algorithm, invariants are perfect to make sure that they are met.

Other obvious JML implementations to consider are implementations for all of the intended methods of the GetIndex- and Build-algorithms as well as the constructor. Since the algorithm makes us of an ArrayList<Node>, a class representation for Node is necessary.

The GetIndex algorithm can be performed in a single method, using the built-in methods of ArrayList and HashMap for lookup and insertion. First off, considering no Node takes on any negative values, no parameters should ever be negative. Another precondition is that since GetIndex makes use of tables T and H, it is necessary to make sure that they are properly initiated. When GetIndex finishes, there are three possibilities. Either parameter two is returned, the index of the already existing node is returned or the index of the new node that is created is returned. Hence the only thing certain about the postcondition is that the method returns an index or one of the last two parameters.

As with the GetIndex algorithm, the Build algorithm can be performed in a single method. The Build algorithm takes two parameters, a Boolean expression *t* and a variable index *i*. For the index variable, a scope between 1 and n+1 is necessary (*k* being the number of variables in the Boolean expression *t*). There cannot be less than one variable in our Boolean expression and if *i* takes on a value greater than n+1 then the Build algorithm has made too many recursive calls rendering the outcome unclear. Another necessary precondition is to make sure that the variable *n* is initialized so that *i* will not be compared with a null value.
Then of course, a precondition is required to make sure that the Boolean expression is a valid. The return value is always 0, 1 or the return value from GetIndex, hence that is also a valid postcondition for Build.

In the constructor for the main class, two things take place. First the number of variables is calculated based on the input from the user. Then the table T is initialized to contain Node 0 and 1. To count the number of variables correctly, a precondition is that the input is correct from the user. The postcondition is that both *n* and T have been correctly initialized.

Finally, the Node constructor initializes its three values to positive numbers, which results in a precondition that all parameters are positive and a postcondition that the initialized variables are positive as well.

## 4 Results

Due to the discovery of some constraints in the Java Modeling language, this section begins with a short introduction of those constraints along with some explanations to why in fact they do exist. This is followed by an analysis for how these constraints affected the implementations done in this paper along with the results of the final implementation of JML.

### 4.1 No Support for Generic Types

The JML2 tool suite, which was used for the verification of the implementations of JML done for this paper, lacks support for generic types. The JML2 tool suite was written and maintained for Java 1.4. Java 1.5, introduced in 2004, brought significant changes to Java, but the work to evolve JML tools to work with Java 1.5 stalled for lack of resources. [6]

The most significant language addition in Java 1.5 was generic types and methods. [6] As Java 1.5 was a rather large upgrade from Java 1.4 this was not the only language feature upgrade. The following features were the language feature upgrades: Generics, Enhanced For Loop, Autoboxing/Unboxing, Typesafe Enums, Varargs, Static import, Metadata. [7] The Enhanced For Loop in this case is the so called for each loop.

## 4.2 Implementation Difficulties

As the project originally was constructed on a runtime environment supporting all the features from java 1.5, early on, this project mainly resulted in JML2 errors.  However, once runtime environment was changed to java 1.4 most errors disappeared. Unfortunately a new error occurred which could not be solved. The use of the methods **get** and **containing** for HashMap resulted in:

**JML2 Error: The actual parameter 1 of method “containing” has type “readonly java.lang.Object” but for this call the method expects a parameter with type “peer java.land.Object”**

Since this error was never solved, the HashMap implementation was replaced by only using ArrayList. The ArrayList was sufficient for this project except it doesn’t execute its method **contains** as fast as HasMap.

## 4.3 Implementation Analysis

The invariants implemented were successful in their task of making sure the uniqueness and non-redundant-test were met. An invariant was added to make sure that the ArrayList T was never empty. When violation against the invariants were made, a JMLruntimeInvariantError occurred and pointed out which method that violated the invariant along with if it did so in the pre- or post-condition of the method call.

```
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLInvariantError: by
method JML.build@pre
```
Figure 11: A runtime exception error from a failed invariant in the precondition of JML.build.

The second invariant states that for all Nodes x, if table T contains x, then that Nodes low value is not equal to its high value. Thus, this second invariant constitutes to the non-redundant-test. The third invariant states that for all Nodes x, if table T contains x, then the index of the last occurrence and first occurrence of x in table T are equal. In other words, no Node x occurs more than once in table T. This corresponds to the uniqueness requirements.

```
/*@
  @ public invariant !T.isEmpty();
  @ public invariant (\forall Node x;
  @ T.contains(x) ==>
  @ x.getLow() != x.getHigh());
  @ public invariant (\forall Node x;
  @ T.contains(x) ==>
  @ T.lastIndexOf(x) == T.indexOf(x));
  @*/
```
Figure 12: The three invariants used in the implementation of the ROBDD algorithm.

As table T, represented by an *ArrayList()*, is needed in the specifications of the invariants, it needed to be made **spec_public** as it was otherwise a private field. Further, the variable *n* representing the number of variables in the Boolean expression, is used in both the constructor and Build method specification and thus also had to be declared **spec_public**. Furthermore, as can be seen in figure 11, the getLow() and getHigh() methods in Node are both used in the specification. Therefore, they had to be declared pure in the method declaration as shown in figure 13:

```
public /*@ pure @*/ int getLow()
```
Figure 13: pure declaration of method getLow() in class Node.

In the constructor there were two preconditions. Those preconditions were intended to make sure that the input Boolean expression was correct. The actually implemented preconditions are by no means complete but they at least make sure that the distribution between variables and operands is correct. The conclusive decision on the validation of the input takes place later on in the Build method. In the constructor, table *T* and variable *n* are both initiated and the postconditions specifies the required states they both need be in after initialization.

```
/*@
@ requires args.length % 2 == 1 &&
@ args[0].equals("x1");
@ ensures !T.isEmpty() && n > 0;
@*/
```
Figure 14: JML specification for the constructor in the main class.

The specifications for GetIndex are quite straightforward. As explained earlier, the only preconditions are that all parameters are positive, and in return the postcondition is positive return value. The variables v, l and h represent the parameters and **result** the return value.

```
/*@ requires v >=  0 && l >= 0 && h >= 0;
  @ ensures \result >= 0;
@*/
```

Figure 15: JML specifications for GetIndex.

As with the GetIndex specification, the specification for the constructor in the class Node corresponded only in a precondition where all parameters needed be positive. The postcondition was to make sure all variables were initialized to their corresponding parameter value. All three field variables had to be declared **spec_public** since they were all part of the constructor specification.

```
/*@ requires a >=  0 && b >= 0 && c >= 0;
  @ ensures a==var && b==low && c==0;
@*/
```

Figure 16: JML specification for constructor in Node.

Each field variable had method getters associated to them. Since they were only to return the value and not change it, there is only one postcondition for these methods. It ensures that the returned value is the same value as the intended return value had before the call was made. They were all identical to the part of the variable name the method represented. The specification for one of these methods is shown in figure 17.

```
/*@
 @  ensures \result == \old(var);
 @*/
```

Figure 17: JML specification for the method getVar() in Node.

The final specification implemented is the specification for the Build method. The Build method takes two parameters, the Boolean expression and an integer *i*. The precondition for *i* is as previously discussed a bound between 1 and n+1. The Boolean expression t is declared to have the same restriction on it as for the input string in the constructor. This is fairly reasonable since the constructor passes the input string to Build, and it is to do so without changing it. The only other precondition is for the number of variables *n* to have been initialized. Finally, the postcondition for Build is that it returns a positive value.

```
/*@ requires 0 < j && j <= n+1;
@ requires n > 0;
@ requires t.length % 2 == 1;
@ ensures \result >= 0;
@*/
```

Figure 18: JML specification for Build.

## 5. Discussion / Conclusions

Working with The Java Modeling Language for the first time was very interesting. Studying for the task of implementing it into actual code made me realize how wide the subject of DBC really is. Unfortunately, in this case, JML is not up to date with the rest of the java language. Several papers have been published on the subject of upgrading JML to at least J2SE5. For now however, if one wishes to work with JML, one would have to do so using J2SE4.

I hope one day it could be integrated into the standard editions of future java development environments in the likes of Eclipse. When working correctly, JML is a joy to work with. For bug prevention and bug detection it can be very useful and in some cases far superior to the tedious task of debugging. Along with pre-and post-conditions, invariants offer a “bigger picture” feeling to source code as it forces the developer to think in a slightly different way. The thought process of developing with JML forces one to take into consideration the relationship between methods and classes more frequently and of major goals of your application. This alone could be great for bug prevention. The fact that JML specifications are not compiled by a regular java compiler further allows it to work as documentation. The syntax is to the point and universal for anyone who knows JML and surely decipherable for any java developer.

What has not yet been brought up in this paper is the fact that JML offers a wide range of tools that stretch far beyond the runtime assertion checker. A very interesting tool is the ESC/Java2 “Extended Static Checker for Java”. Its main goal is to find runtime errors at compile time. Future studies could be done on this or any other of the interesting tools offered for the Java Modeling Language.

## Bibliography

Gary T. Leavens and Yoonsik Cheon (2006), *Design by Contract with JML*.
http://www.eecs.ucf.edu/~leavens/JML//jmldbc.pdf [1]

Henrik Reif Andersen (1999), *An Introduction to Binary Decision Diagrams*
http://www.itu.dk/courses/EAP/Notes/bdd-eap.pdf [2]

M. Huth, M. Ryan , (2004),
*Logic in Computer Science 2nd ed* [3]

Daniel Bruns (2009), *Formal Semantics for the Java Modeling Language*.
http://lfm.iti.uni-karlsruhe.de/download/Diplomarbeit_DanielBruns.pdf [4]

Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens and Erik Poll,
*Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*
http://www.eecs.ucf.edu/~leavens/JML/fmco.pdf [5]

David R.Cock, *Adapting JML to generic types and Java 1.6*
Eastman Kodak Company Research Laboratory
1999 Lake Avenue
Rochester, NY 14650 USA [6]

*New Features and Enhancements J2SE 5.0*
http://download.oracle.com/javase/1.5.0/docs/relnotes/features.html#boxing [7]

*Getting started with JML, improve your Java programs with JML annotation*. [8]
http://www.ibm.com/developerworks/java/library/j-jml/index.html