Abstract

Modern software development projects are extremely complex and often involve millions of lines of code. Using the Java Modeling Language (JML) can substantially reduce bugs and errors in software implemented in Java. The JML tool from Iowa State has many features, including static checking and run-time assertion checking of preconditions. However, the run-time assertion checking lacks the ability to check quantified expressions in JML. This paper describes the extension of the assertion checker to support quantified expressions.
I. Introduction

Most of today’s software applications are extremely complex and often involve millions of lines of code. For example, Microsoft Windows 2000 and Microsoft Office 2001 (Mac) are reported to contain about 30 million lines of code each [Rea00][Spa00]. It is therefore not surprising that controlling the number of software glitches and bugs is a difficult task, even through intensive software testing. In fact, testing only determines that the specific tests used revealed no bugs but does not guarantee that there are no bugs in the software.

With non-critical applications, tracking down bugs by extensive testing and re-testing is a very practical solution. However, when the software produced is used to run nuclear power plants or operate medical devices, simply executing a set of tests may not be enough to assure the software would not malfunction and cause a catastrophe. Clearly, there needs to be a way to ensure that the software meets the precise specification given for its functionality.

Software specifications given through human languages such as English are not ideal solutions since human languages themselves are inherently ambiguous. In order to assure that software strictly adheres to its specifications, there needs to be a precise and unambiguous way of specifying the functionality of software. Such a specification is called a formal specification.

Defining software functionality using formal specifications has several advantages. The main advantages include [Wah01]:

1. The software behavior is defined in a precise and unambiguous manner so that the functionality of the software can be communicated effectively between programmers implementing different parts of the software system.

2. It may be possible to provide detailed analyses and proofs of correctness of the software system.
3. Executable specification languages can be used to verify that the software operates within
   the specifications or to generate software prototypes.
4. Formal specifications can serve as guides for producing test cases.

   It can easily be seen that the advantages gained using formal specifications can be great.
   The ability to provide precise specification of software behavior can help reduce the amount of
testing and debugging significantly. Formal proofs of correctness further assure the clients that
the software complies with the specification.

   This project extends the functionality of the software from Iowa State that supports Java
Modeling Language specifications. The current tool does not support assertion checking for
quantified expressions. The added functionality provides support for three simple forms of
quantified expressions in the runtime assertion checker. These forms of expressions are
commonly used in specifications and therefore greatly increases the usefulness of the tool.

**II. The Java Modeling Language**

   The Java Modeling Language (JML) [LBR01] is a model-based formal specification
language designed for Java. A model-based specification is defined using familiar mathematical
data types such as sets, sequences, and functions as well as primitive types such as integers and
characters [Wah01]. Pre- and post-conditions are then defined over the model data types based
on a suggested model of the implementation. Because JML is used to document the behavior
and interface of Java modules, it is also known as a *behavioral interface specification language*.

   JML has been designed to be easy for Java programmers to learn and understand.
Logical *AND* (&&) and *OR* (||) operators, for example, are represented the same way in JML as
in Java. JML has also been designed to document existing Java software without limitations on
the design methods used to create it.

Specifications in JML are typically included with code as C-style comments (/* ... */) or as C++-style (// ... ) comments with a special at-sign (@) to indicate the comment as a JML specification. JML specifications can also be given in separate files from the source code.

Because JML behavioral specifications are comments, they are ignored by the Java compiler.

When in the form of C-style comments, the at-sign at the beginning and end of the comments indicates that the comment block is a JML specification. In C++-style comments however, only the at-sign at the beginning of the line marks the comment as a JML specification. In C-style comments, optional at-signs can also be added to the beginning of each line to help make the specification more legible. Figure 1 is a simple example of a JML specification with line numbers on the right-hand side as C++-style comments for easier reference. These comments are not used by JML or Java.

```
public class IntMathOps { // 1
   /*@ public normal_behavior // 2
      @   requires y >= 0; // 3
      @   ensures \result * \result <= y // 4
      @     && y < (Math.abs(\result) + 1) // 5
      @     * (Math.abs(\result) + 1); // 6
      @*/ // 7
   public static int isqrt(int y) { // 8
      return (int) Math.sqrt(y); // 9
   } // 10
} // 11
```

**Figure 1: class IntMathOps (Adapted from [LBR01])**

The specification in Figure 1 describes the class IntMathOps with a public static method isqrt. The declared interface is composed of line 1 and line 9. The behaviors specified are described in the C-style comments from lines 3 to 8. In the specification, the keyword public indicates that the specification is intended for clients. The keyword normal_behavior indicates that the method call will return normally and throw no
exceptions if the precondition is met. Pre- and post-conditions are designated with the keywords requires and ensures, respectively.

JML specifications are commonly written as comments in source files. However, as previously mentioned, it is also possible to write JML specification files separately from the source file. Such files do not contain source code and contain only the class and method declarations. Method declarations are terminated with a semicolon (;) as in Java abstract methods. To refer to the JML specification file, the source file should include a single JML refine statement at the top of the file. Figure 2 and Figure 3 show the contents of the files IntMathOps.jml and IntMathOps.java, containing the JML specification and the Java source, respectively. Line numbers have been omitted in both files.

```java
public class IntMathOps {
    //@ public normal_behavior
    @   requires y >= 0;
    @   ensures \result * \result <= y
         && y < (Math.abs(\result) + 1) * (Math.abs(\result) + 1);
    @*/
    public static int isqrt(int y);
}

Figure 2: IntMathOps.jml
//@ refine IntMathOps <- "IntMathOps.jml";
public class IntMathOps {
    public static int isqrt(int y) {
        return (int) Math.sqrt(y);
    }
}

Figure 3: IntMathOps.java with a refine statement [LBR01]

JML allows specifications to be written at any level of detail desired. For example, in the specification for IntMathOps, we can specify just that the argument passed to the method must be a positive integer. This results in the following:
public class IntMathOps {
    /*@ requires y >= 0;
    @*/
    public static int isqrt(int y) {
        return (int) Math.sqrt(y);
    }
}

Figure 4: A partial specification of IntMathOps [LBR01]

When the public keyword is removed in this case, it results in the specification being implicitly public. Removing the keyword normal_behavior denotes that exceptions may be thrown by the method, or it may terminate normally. Having no ensures clause indicates that the value returned is not restricted in any way. The default meanings of various clauses are listed in Appendix A of “Preliminary Design of JML” [LBR01].

JML has an extensive set of features as well as a large grammar. For this paper, only syntax related to the project will be discussed. The following is a brief overview of basic JML specification syntax.

The assignable keyword permits the specified object to be modified. In JML, as with Java, objects are only accessed through reference types. Therefore, technically we really mean that the object specified is permitted to be modified. Keywords modifies and modifiable can also be used instead.

The ensures keyword refers to the postcondition of the method, which is the condition that must be true if the precondition has been met. Similarly, the requires keyword refers to the precondition, which is the condition that must hold before the method is called.

The invariant keyword specifies the conditions that must be preserved for all publicly visible instances of the class. Invariants do not have to hold while executing an object’s methods but should hold before and after execution.

The depends and represents keywords show the relationships between model and
concrete variables. Model variables are data fields used only in the JML specification whereas concrete variables are data fields declared as part of the implementation. The keyword depends indicates the dependencies of concrete variables on the model variables. The representation of values of concrete variables in relation to the model variables is specified using the represents keyword. The represents statement can only be used to state the relationship between dependents and dependees.

When the initially keyword is attached to a field, it sets the initial value of the field. That is, when the instance is created all reachable objects will observe the same initial value of the field.

JML supports the use of $\forall$ and $\exists$ for universal and existential quantifiers. Several useful extensions to quantified expressions have also been added such as $\max$, $\min$, $\product$, and $\sum$. Minimum and maximum values of a set can be expressed using $\min$ and $\max$. Similarly, the products and sums of a set of values can be expressed using $\product$ and $\sum$. The syntax for the quantifiers will be discussed later in this paper.

### III. The JML Tool

There are three tools for JML that the author is aware of. They are ESC/Java [LLN00], LOOP [JBH98], and the JML release from Iowa State. This paper focuses on assertion checking in Iowa State’s JML release and thus will not refer to the other tools.

A tool called ANTLR [Par99] is used to construct much of the JML tool. ANTLR is an acronym for “ANother Tool for Language Recognition” and is used for creating compilers. ANTLR can be used with C++ or Java and supports many features. Unlike the traditional tools lex and yacc which are used for creating compilers, ANTLR incorporates the lexical analyzer
The JML tool supports static checking of specifications, HTML documentation generation, and runtime assertion checking. Static checking checks the syntax and types of the expressions in JML specifications as well as Java source code. The documentation generator generates web pages of JML specification in a format similar to javadoc [Fla99], the Java API documentation generator. The assertion checking feature supports precondition, postcondition, and as invariant checking.

The JML tool can be run from the Windows command prompt or from a shell prompt in a Unix environment. Manual pages document numerous command-line options that can be passed, including options to output the abstract syntax tree, generate documentation in HTML, and generate assertion checking code.

The JML tool is still under development and has many limitations. The assertion checking feature, for example, only supports simple logical expressions that do not include quantifiers. The assertion checker is also unable to generate code for JML specifications that do not contain boolean subexpressions.

The inability to execute assertions in predicates with quantifiers is a major deficiency in the JML tool. Quantifiers are powerful and useful constructs for handling a large number of logical propositions. As a result, they are often used in formal specifications. The goal of this project is to extend the runtime assertion checking to handle quantified expressions in JML. Because only precondition checking was available during this project, our work only deals with quantifiers in the precondition. We will now examine the implementation of the runtime assertion checking tool in the JML tool and the modifications necessary in order to support
IV. The JML Assertion Checker

The JML assertion checker tool is composed of two components. The first component is the translator that converts the original source with JML specifications to another source file with the same name that includes the assertion checking code. The second component of the tool is the runtime system which allows the user to specify the various assertion checking options. The translator is integrated with the JML type checker and manipulates the abstract syntax tree (AST) associated with the source code to generate assertion checking code. All of the original source code is retained and thus the functionality of the class is preserved.

We will examine an example of the translation done by the assertion checker. The same IntMathOps example previously used is run through the JML tool resulting in the following:
public class IntMathOps
{
    /*@
     @   public normal_behavior
     @     requires y >= 0;
     @     ensures \result * \result <= y &&
     @         y < ((Math.abs(\result)) + 1) * ((Math.abs(\result)) + 1);
     @*/
    public static int isqrt(int y)
    {
        if (edu.iastate.cs.jml.checker.runtime.Checker.isActive(
            edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION))
        {
            edu.iastate.cs.jml.checker.runtime.Checker.enterAssertionCheck();
            boolean __jml0;
            try {
                __jml0 = y >= 0;
            } catch (Exception e_xyzzy50010) {
                __jml0 = false;
            }
            edu.iastate.cs.jml.checker.runtime.Checker.assert(__jml0,
                edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION,
                "IntMathOps.java", 10, "IntMathOps", "isqrt", null);
            edu.iastate.cs.jml.checker.runtime.Checker.exitAssertionCheck();
        }
        return (int)(Math.sqrt(y));
    }
}

Figure 5: IntMathOps.java (generated source code)
Formatting of the output code above has been altered to accommodate the page width.

In the generated code, an if block is generated to allow the user to specify whether to execute the precondition assertion code at runtime. The methods enterAssertionCheck() and exitAssertionCheck() are used to detect possible recursions that might occur by setting a flag in the symbol table. In between the two method calls is the actual code that checks the assertion.

First, a boolean variable is declared that holds the result of the assertion. In this case, the only condition that needs to be checked is if the variable y is greater than or equal to 0 (on line 4 of Figure 1). The boolean value of the condition is then assigned to the boolean variable declared earlier and it is then wrapped inside a try-catch block. The expression in this example would not throw any exceptions. However, if a more complicated mathematical expression were
used, exceptions (such as divide by zero) could be thrown where the boolean result would simply be set to false.

The method assert() is called to check the value of the boolean result variable. If the assertion fails, an error message is displayed that shows the class, method name, and line number in the original source code where the assertion failed.

This example demonstrates the type of predicates handled by the assertion checker. As mentioned, if a quantified expression exists in the specification, assertion checking for that method would be disabled in the original version of the JML tool. A limited number of forms of quantified expressions were added to extend the capability of the assertion checker. The following looks at the different forms of quantified expressions that are now supported by the assertion checker.

V. Executable Quantified Expressions

The variable bound by a quantifier can range over either a finite or infinite domain of values which may or may not be contiguous. This poses a difficult task of retrieving the domain set for iteration during assertion checking. In general, a quantified expression follows this syntax:

\[(quantifier \text{ type } \text{ var}; \text{ constraint}_\text{ expr}; \text{ expr})\]

The quantifier in the above syntax denotes the name of the quantifier, such as \(\forall, \exists, \sum, \text{ or } \prod\). Following the keyword is the variable declaration for the variable bound by the quantifier. Semicolons (;) delimit the declaration, the constraint expression, and the body of the expression (expr). The constraint expression is optional.

For simplicity, the supported quantified expressions that we have added are limited to
expressions with the following formats (note that all contain a constraint expression):

1. \((\text{quantifier } rtype \ i; \ obj\ .\ contains(i); \ expr)\)
2. \((\text{quantifier } rtype \ i; \ obj\ .\ has(i); \ expr)\)
3. \((\text{quantifier } ptype \ i; \ x \leq i \ \&\& \ i < y; \ expr)\)

In the above list, \textit{quantifier} denotes either the existential or the universal quantifier (\exists and \forall). The \textit{rtype} in the first two formats denotes a reference type and \textit{ptype} denotes one of the integral numeric types in Java (\texttt{int}, \texttt{short}, \texttt{long}, \texttt{byte}, \texttt{char}). The \(i\) denotes any valid variable name and \textit{obj} denotes a reference variable. In the third format, \(x\) and \(y\) denote variables or arithmetic expressions. Finally, \textit{expr} denotes a boolean expression. Both \texttt{contains()} and \texttt{has()} are methods of the specified object.

In the first format, it is possible that the object is an instance of \texttt{Hashtable} or \texttt{Vector} from the \texttt{java.util} package. In such a case, we can retrieve the domain by using an \texttt{Enumeration} over either of these classes. For Java 1.2 and greater, however, it is also possible that the object may have implemented the \texttt{Collection} interface. This interface allows an \texttt{Iterator} to be returned which we can use to retrieve the domain. In the second format, we check objects for instances of the JML model classes that utilize \texttt{Enumeration} (\texttt{JMOBJBag}, \texttt{JMOBJSequence}, \texttt{JMOBJSet}, \texttt{JMValueBag}, \texttt{JMValueSet}, and \texttt{JMValueSequence}). The domain can then be retrieved in the same manner as the first format for \texttt{Hashtable} and \texttt{Vector}. In the third format, we use a common form of specifying a range of numbers. Retrieving the domain for the third form is therefore, trivial.

Other quantifiers such as \(\Sigma\), \(\Pi\), \(\max\), and \(\min\) can easily be supported in the future using the third format (with \textit{expr} as a numeric type). However, this requires an
extensive modification of the assertion checker implementation. The assertion checker does not properly support subexpressions that are not of type boolean. Therefore, because $\sum$, $\prod$, $\max$, and $\min$ are numeric expressions, they are not yet supported.

Because the formats of the quantified expressions supported are precisely defined, identifying executable quantified expressions is a simple task. A quantified expression can be identified by traversing its abstract syntax tree (AST) and checking that the branches and nodes of the tree match the specified format. Upon determining the quantifier type and format, the AST of the generated code is created and attached to AST of the source code.

VI. Assertion Code Generation

As mentioned, the general form of the quantified expression assertion checking code involves iterating over the associated collection of values, retrieving those values in the domain, and testing the boolean expression associated with the quantifier over the retrieved values. The following describes the generated code in detail for each of the supported forms of quantified expressions.

For the first quantified expression form involving a `contains()` method in the constraint, there are three possibilities to consider. The first is that the object that has a `contains()` method is an instance of `java.util.Vector` or `java.util.Hashtable`. The generated code tests for this condition at runtime using the `instanceof` operator. The body of the `if` statement (for when this condition is true) retrieves the `Enumeration` for the collection and iterates through each object in the collection using a `while` loop. Each associated object extracted from the collection is then tested against `expr` in the quantified expression. The results of each test are then accumulated using the logical AND (`&&`) operator if it is a
\forall expression, or logical OR (| |) if it is an \exists expression.

The second possibility for the first format is that the object in the constraint implements the java.util.Collection interface (only applies to Java 1.2 or later). Another if statement in the body of the else for the previous if-statement tests for this condition. If this condition tests true, an Iterator is retrieved from the collection object. A while loop similar to the previous case is used to retrieve the objects in the collection and accumulate the results of evaluating the expression expr though each iteration of the loop.

The third possibility is that the contains() method of the object in the constraint is a user-defined method. In such a case, we cannot assume anything about the object class and therefore would not be able to generate assertion checking code. Therefore, in the body of the else of the above if-statement, we only generate a warning message to the user that the assertion is not checked, even though the quantified expression is in a supported format.

For quantified expressions in the second format involving the has() method in the constraint, we use the instanceof operator to check if the object is an instance of a model class that contains the method for retrieving a java.util.Enumeration object. As mentioned previously, the model classes that use Enumeration are JMLObjectBag, JMLObjectSequence, JMLObjectSet, JMLValueBag, JMLValueSet, and JMLValueSequence. The assertion checking code for this format is identical to the first format where Enumeration is also used.

It is important to note that the current assertion checker does not support expressions that refer to model variables. Thus, specifications that refer to model variables would generate no assertion-checking code. Nevertheless, assertion checking can still be used in cases where the JML classes indicated above are employed in an implementation.
In the third form of quantified expression where a numerical range is indicated in the constraint, the assertion checking code is simple. Assertion checking code for this format is a for loop, with the initial value of the loop variable set to the lower bound of the numerical range and the loop repetition condition equivalent to true while the loop variable is less than the upper bound of the range. The loop update of the for loop increments the loop variable by one. The expr expression is then tested by applying each value in the domain, accumulating the results with the appropriate logical AND (&&) or OR (||) operator.

We will now demonstrate the output of the generated code using two examples. First we will show a specification for an ADT followed by an implementation of the specification. Then we will show the generated assertion code for the two methods with supported quantified expressions in their precondition.

Suppose we have a JML specification for a simple list ADT MyList in the file MyList.jml that contains the following:
This list class contains only the operations `MyList()`, `add()`, `sum()`, and `contains()` for simplicity. It can be easily seen that this class can only add objects of class `Integer`. However, because this specification does not indicate the constraints on the type of objects in the list, a valid list may also contain objects not of class `Integer`. It is possible, for example, to have some complex operation of the ADT that modifies list such that one or more objects in the list are not an `Integer` objects. For this example, we will assume such a possibility and that the `sum()` operation will only return an object `Integer` that contains the
sum of the list of integers if all the objects are of class `Integer`. The behavior of the `sum()` method is undefined if one or more objects are not of class `Integer`.

Next we will present an implementation of the above specification in `MyList.java` file. This file will not only contain the code that implements the ADT but also the JML specification of the implementation. The implementation code is as follows:
//@ refine MyList <- "MyList.jml";
//@ model import edu.iastate.cs.jml.models.*;

import java.util.*;

public class MyList {
    //@ private depends theList<-list;
    //@ private represents theList \such_that
    @  theList.length() == list.size() 
    @  (\forall int i; 0 <= i && i < theList.length();
    @    list.elementAt(i) == theList.itemAt(i) );
    @*/

    private Vector list;

    //@ private normal_behavior
    @ assignable list;
    @ ensures list != null && list.size() == 0;
    @*/
    public MyList() { list = new Vector(); }

    //@ private normal_behavior
    @ requires (\forall int j; 0 <= j && j < list.size();
    @     list.elementAt(j) != i);
    @ assignable list;
    @ ensures list.size() == \old(list.size()) + 1 &&
    @     (\forall int j; 0 <= j && j < \old(list.size());
    @     list.elementAt(j) == \old(list.elementAt(j)) &&
    @     list.lastElement() == i;)
    @*/
    public void add(Integer i) { list.add(i); }

    //@ private normal_behavior
    @ requires (\forall Object o; list.contains(o); o instanceof Integer);
    @ ensures \result != null &&
    @     \result == new Integer(
    @     (\sum int i;
    @     (\exists Integer j; list.contains(j); j.intValue() == i);
    @     i ));
    @*/
    public Integer sum() {
        int totalsum = 0;
        for (int i = 0; i < list.size(); i++)
            totalsum += ((Integer) list.get(i)).intValue();
        return new Integer(totalsum);
    }

    //@ private normal_behavior
    @ ensures \result ==
    @     (\exists int i; 0 <= i && i < list.size(); list.elementAt(i) == o);
    @*/
    public boolean contains(Object o) {
        boolean res = false;
        return res;
for (int i = 0; i < list.size(); i++)
    if (o == list.elementAt(i)) {
        res = true;
        break;
    }
    return res;
}

Figure 7: MyList.java

This implementation of the ADT adheres closely to the specification, using Vector in place of JMLObjectSequence. Instances of Integer are then added to list through the add() method. For the sum() operation, the precondition is the same as the precondition in MyList.jml (Figure 6), and requires that all objects in list be instances of Integer. The contains() operation is different from the contains operation of class Vector, which is why the result of the contains() method of Vector cannot be returned in this case. In class Vector, the contains() operation returns true if an object in the vector is equal using the equals() operation. The has() method of JMLObjectSequence, however, is true only if the same object is in the sequence (using the == operator).

It can be seen that the JML specification of the implementation is quite similar to the ADT specification. Typically, proofs would be done at this point to show that the implementation satisfies the specification. However, we will not do that here as that is beyond the scope of this paper.

We will now look at the sum() operation of MyList and examine the generated assertion code for the precondition. The precondition of the sum() is a quantifier with a contains() method in the constraint and hence matches the first of the three supported forms quantifiers. Running the JML tool with assertion checking results in the following output for the sum() operation:
/**
 * @private normal_behavior
 * @requires (\forall Object o; list.contains(o); o instanceof Integer );
 * @ensures \result != null &&
 *   \result ==
 *     new Integer((\sum int i; (\exists Integer j; list.contains(j);
 *       (j.intValue()) == i); i));
 */

public Integer sum()
{
    if (edu.iastate.cs.jml.checker.runtime.Checker
        .isActive(edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION))
    {
        edu.iastate.cs.jml.checker.runtime.Checker.enterAssertionCheck();
        boolean __jml0;
        try
        {
            boolean __jml1 = true;
            try
            {
                boolean __jml3 = true;
                try
                {
                    __jml3 = o instanceof Integer;
                }
                catch (Exception e_xyzzy50010)
                {
                    __jml3 = false;
                }
                __jml1 = __jml1 && __jml3;
            }
            while (__jml1 && (__jml2.hasMoreElements()))
            {
                o = (Object)(__jml2.nextElement());
                boolean __jml3;
                try
                {
                    __jml3 = o instanceof Integer;
                }
                catch (Exception e_xyzzy50010)
                {
                    __jml3 = false;
                }
                __jml1 = __jml1 && __jml3;
            }
        }
        else
        {
            boolean __jml3 = true;
            try
            {
                __jml3 = o instanceof Integer;
            }
            catch (Exception e_xyzzy50010)
            {
                __jml3 = false;
            }
            __jml1 = __jml1 && __jml3;
        }
    }
    return super.sum();
}
As mentioned, the precondition assertion code for the quantified expression with the
contains() constraint involves more than one step. First a test is needed to check if the
object in the constraint field of the quantified expression is a known collection class that uses
java.utilEnumeration. In the standard Java library, only Vector and Hashtable
from the java.util package utilize Enumeration.

If this check succeeds, the relevant assertion code is then executed to check the
precondition. In this case, the method `elements()` is invoked to retrieve an `Enumeration` from the collection object. This is done indirectly through multiple casts and using the `getClass()`, `getMethod()` and `invoke()` methods in order to resolve type checking issues. A `while` loop is used to retrieve each object in the collection and an assertion check is then done on each retrieved object. Depending on the type of quantifier (`\forall` or `\exists`), the appropriate logical operator (logical AND in this case) is then used to accumulate the results.

If the check against `Vector` and `Hashtable` fails, we will check to see if the object implements the `java.util.Collection` interface. If the check succeeds, we assume that the appropriate code is available to retrieve an `Iterator` which we can use to retrieve each object in the collection as the case with `Enumeration`. It is also possible for the check against `java.util.Collection` to fail. As mentioned before, the user in this case has implemented his own method using the same name (as is the case with our `MyList` class). For this situation, we print a warning indicating the class, method, and line number, and that no checking was done.

Assertion checking code for the third form (where we have a numerical range in the constraint) can also be seen in the `MyList` example in Figure 7. For the `add()` operation, we have a precondition with a quantified expression that has a numerical range in the constraint. The output of the assertion checking code for `add()` is shown in Figure 9:
/*@ private normal_behavior
 @ requires (\forall int j; 0 <= j && j < (list.size());
 (list.elementAt(j)) != i);
 @ assignable list;
 @ ensures ((list.size()) == \old(list.size()) + 1 &&
 (\forall int j; 0 <= j && j < \old(list.size()); (list.elementAt(j))
 == \old(list.elementAt(j)))) && (list.lastElement()) == j;
 @*/

public void add(Integer i)
{
    if (edu.iastate.cs.jml.checker.runtime.Checker
        .isActive(edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION))
    {
        edu.iastate.cs.jml.checker.runtime.Checker.enterAssertionCheck();
        boolean __jml0;
        try
        {
            boolean __jml1 = true;
            try
            {
                for (int j = (int )(Math.ceil(0)); j < (list.size()); j++)
                {
                    boolean __jml2;
                    try
                    {
                        __jml2 = (list.elementAt(j)) != i;
                    }
                    catch (Exception e_xyzzy50010)
                    {
                        __jml2 = false;
                    }
                    __jml1 = __jml1 && __jml2;
                }
                catch (Exception e_xyzzy50010)
                {
                    __jml1 = false;
                    __jml0 = __jml1;
                }
                catch (Exception e_xyzzy50010)
                {
                    __jml0 = false;
                }
                edu.iastate.cs.jml.checker.runtime.Checker.assert(__jml0,
                    edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION,
                    "MyList.java", 30, "MyList", "add", null);
            }
            edu.iastate.cs.jml.checker.runtime.Checker.exitAssertionCheck();
        }
    }
    list.add(i);
}

Figure 9: Generated assertion checking code for add()
From Figure 9, we can see that the assertion checking code for `add()` consists of a simple `for` loop. The initial value of the `for` loop variable is set to the ceiling of the lower bound of the numeric expression and cast back to the integral data type used in the bound variable declaration in the quantified expression. This procedure is necessary because the lower bound may result in an expression of type `float` or `double`. Results of the `expr` expression in the loop are accumulated the same way as before, using the appropriate logical operator to accumulate the results in each loop iteration.

VII. Future Work

Although support for commonly used forms of quantified expressions is now available, support for more forms of quantified expressions can substantially enhance the JML tool. For example, more flexibility can be added for the syntax of the range of integral values. Support for expressions utilizing `\max`, `\min`, `\product`, and `\sum` has been added but is not yet usable. Enabling the use of these quantifiers requires the ability to extract type information from the JML specification and thus modification to the current assertion checker system is necessary.

A helper class in the JML tool could assist in generating ASTs from valid Java code fragments. Currently, ASTs for generated code are manually coded into the JML tool, which can be tedious for large chunks of Java code. It would be useful to construct Java code as a string and run the string through a parser that returns an AST. Furthermore, modifications in the tree structure in the future would have little or no impact on the assertion checker if such a helper class were used.

Finally, because invariant and postcondition assertion checking is now available in the JML tool, the next step would be to add support for quantifiers in postcondition assertion
checking. Invariants often contain quantified expressions and the work here can also be applied to invariant assertion checking.

VIII. Conclusion

Most of the software systems developed today are extremely complicated, resulting in a large number of undetected bugs despite rigorous testing. The use of formal specifications provides a precise way of specifying software behavior, helping to reduce and detect bugs in software. JML is a model-based formal specification language designed for the Java programming language. For critical software systems, JML is vital for defining the exact behavior of Java software.

The JML tool from Iowa State University provides various features and functions supporting JML. It supports static checking of JML, generating HTML documentation for JML specifications, and runtime assertion checking.

The assertion checker of the JML tool is important for software validation. However, the inability to execute quantified expressions for assertion checking is a critical deficiency. Quantified expressions are powerful constructs for specifying predicates on collections of items and are frequently used. To improve the JML assertion checker, support for three commonly used forms of quantified expressions has been added. The added support for the three forms greatly increases the versatility and usefulness of the JML tool.
References

Bho00  Bhorkar, A.  A Run-time Assertion Checker for Java using JML.  Technical Report TR00-08, Department of Computer Science, Iowa State University, Ames, Iowa 50011.  May 2000.


