Implementation of Refining Statements in OpenJML and Verification of Higher Order Methods with Model Program Specifications

2017

Sai Chandesh Gurramkonda

University of Central Florida

IMPLEMENTATION OF REFINING STATEMENTS IN OPENJML AND VERIFICATION OF HIGHER ORDER METHODS WITH MODEL PROGRAM SPECIFICATIONS

by

SAI CHANDESH GURRAMKONDA
B.S. Jawaharlal Nehru Technological University, 2013

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Summer Term
2017

ABSTRACT

The Java Modeling Language (JML) describes the functional behavior of Java classes and methods using pre- and postconditions. However, standard pre- and postcondition specifications cannot verify calls to higher order methods (HOMs). JML uses model program specifications to reason about HOMs. This thesis describes the implementation of model programs in the OpenJML tool. The implementation includes parsing, type checking, and matching of model program specifications against the code.

I dedicate this thesis to my parents G.Jagadeeswara Prasad and G.Ambuja, for all of the sacrifices you made to ensure my success. Thank you for supporting me in all my endeavors and teaching me the value of hard work.

I would also like to dedicate this thesis to my sisters G.Sai Prasanna, M.Reva Madhavi and my brother-in-law K.Bharat Kumar, who have been a constant source of support and encouragement during the challenges of graduate school and life.

ACKNOWLEDGMENTS

I would like to express the deepest appreciation to my advisor, professor Dr. Gary T.Leavens, who has the attitude and the substance of a genius. Without his encouragement, advice and persistent helps, this thesis would not have been possible. I appreciate all his contributions of time, patience and ideas to make my thesis experience productive. I would also like to thank my thesis committee members, professors Dr. Sumit Kumar Jha and Dr. Damla Turgut for their guidance during my thesis work. I would also like to thank Dr.Rochelle Elva for her support during my course of study.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

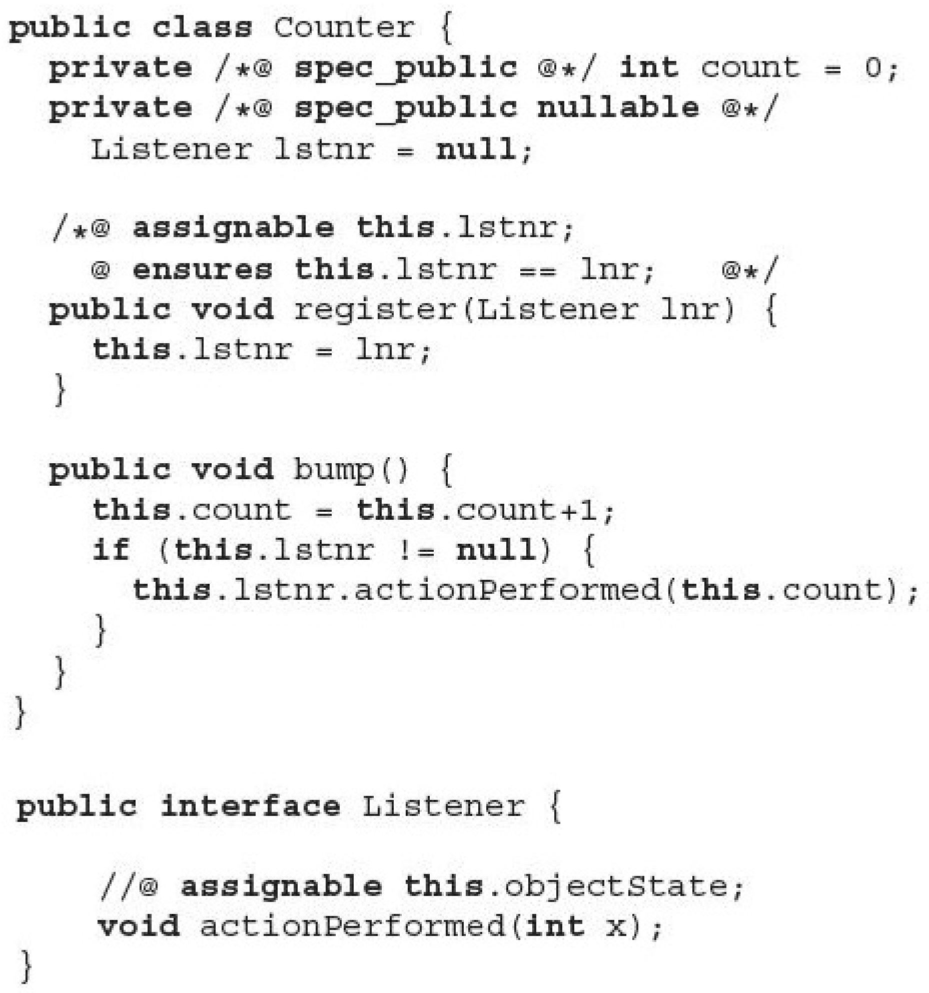
The Java Modeling Language (JML) is a behavioral interface specific language (BISL) [2] which is designed to specify Java modules. It is based on Design by Contract (DBC) approach, which was proposed by Bertrand Meyer and also based on the model-based specification approach of Larch family of interface specification languages [17] and the refinement calculus [16]. The main 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. The use of such pre- and postconditions to specify software dates back to Hoare's paper on formal verification [14]. These pre- and postconditions are expressed as structured Java comments or Java annotations that use Java-like logical expressions by using OpenJML - a tool set for JML, built on the OpenJDK framework for Java. But standard pre- and postconditions cannot verify calls to higher order methods (HOMs) - methods which makes mandatory calls using dynamic dispatch.

So in JML, model program specifications are used to reason about HOMs. The model program specifications are described in Shaner, Leavens and Naumann's paper [1]. Model programs are used to write specifications of HOMs, which exposes information about the method's mandatory calls, while hiding some details. This is achieved by checking that HOMs implement a model program specification, using refining statements. But a practical implementation of refining statements and verification of calls to HOMs in OpenJML does not exist. This thesis will explain the implementation of refining statements in OpenJML and first step of verification of calls to HOMs.

The Problem

The practical implementation of refining statements and verification of calls to HOMs are existing problems in OpenJML. Implementation of refining statements is very important as it helps in reasoning about the HOMs, which are the methods that makes calls to methods that are either dynamically dispatched or have weak specifications. Reasoning about HOMs through refining statements is done by dynamically checking that HOMs implement model program specifications.

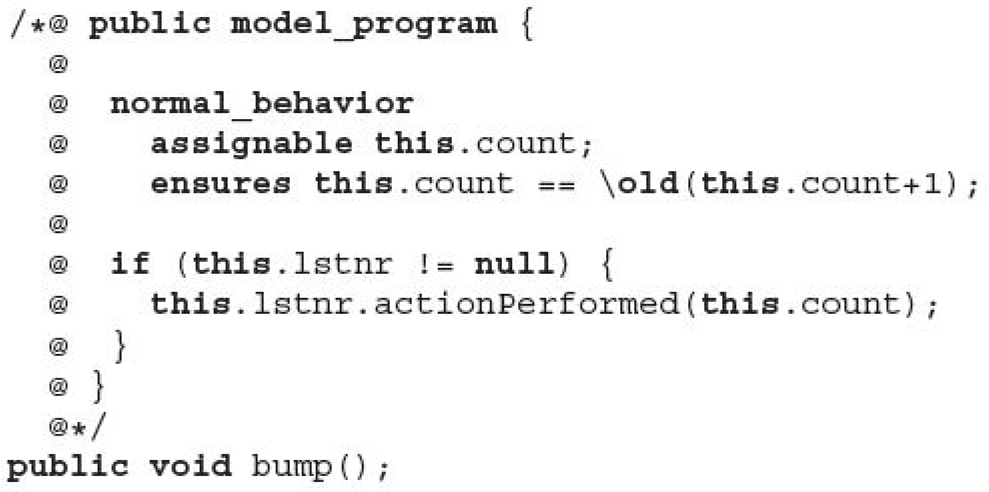
To understand what is involved in the implementation of JML model programs, consider an HOM example as noted in Shaner et al [1, p.2] : "As shown in Figure 1.1, the `bump()` method in the `Counter` class makes a mandatory call to the `actionPerformed()` method in the `Listener()` interface. The specifications of the `actionPerformed()` method are weak because it has no pre- and postconditions. The only constraint on its actions is given by the specification's assignable clause. This clause names `this.objectState`, which is a datagroup defined for class `Object`. A datagroup is a declared set of fields that can be added to in subtypes [15], thus making `bump()` method as a HOM. To reason about the `bump()` method, model program specifications are written as shown in Figure 1.2. The keyword `model_program` introduces the model program. Its body contains a statement sequence consisting of a specification statement followed by an if-statement. The specification statement starts with `normal_behavior` and includes the assignable and ensures clauses. Specification statements can also have a requires clause, which would give a precondition; in this example the precondition defaults to `true`. A specification statement describes the effect of a piece of code that would be used at that place in an implementation. Such a piece of code can assume the specification statement's precondition and must establish its postcondition, assigning only to the datagroups permitted by its assignable clause. Thus specification statements can hide implementation details and make the model program more abstract. Although the example uses a specification statement in a trivial way, they can be used to abstract arbitrary pieces of code, and have been used to do so in the refinement calculus [16].”



```
public class Counter {
  private /*@ spec_public @*/ int count = 0;
  private /*@ spec_public nullable @*/
    Listener lstnr = null;

  /*@ assignable this.lstnr;
    @ ensures this.lstnr == lnr;   @*/
  public void register(Listener lnr) {
    this.lstnr = lnr;
  }

  public void bump() {
    this.count = this.count+1;
    if (this.lstnr != null) {
      this.lstnr.actionPerformed(this.count);
    }
  }
}

public interface Listener {

    //@ assignable this.objectState;
    void actionPerformed(int x);
}
```

Figure 1.1: Java Class with JML specifications (quoted from [1])

In the implementation of model programs[1], there are two steps after parsing: matching and verification of refining statements. The first step in the implementation is matching the code against the model program, which yields a set of verification conditions for parts of the code that implement the model program's specification statements and also whether the code has the form specified by the model program. The matching is done by verifying that the code is exactly matched with the model program except the specification statements. The specification statements in the model program are matched against the refining statements in the code. In the example shown in figure 1.3,the `bump()`'s method code refining statements match the model program specifications in the figure 1.2 and the code that calls to `actionPerformed()` method matches the code in the model program. Therefore, the `bump()` method code matches the model program. The second step in the implementation is verification of refining statements which is done by proving that the refining statement implements its specifications.



```
/*@ public model_program {
  @
  @  normal_behavior
  @    assignable this.count;
  @    ensures this.count == \old(this.count+1);
  @
  @  if (this.lstnr != null) {
  @    this.lstnr.actionPerformed(this.count);
  @  }
  @ }
  @*/
public void bump();
```

Figure 1.2: Model Program for the `bump()` method (quoted from [1])

[1]This discussion is adapted from Shaner et al's work[1, p.5]

```
public void bump() {
  /*@ refining normal_behavior
    @   assignable this.count;
    @   ensures this.count == \old(this.count+1);
    @*/
  this.count = this.count+1;

  if (this.lstnr != null) {
    this.lstnr.actionPerformed(this.count);
  }
}
```

Figure 1.3: Code matching model program specification (quoted from [1])

Related Work

Shaner, Leavens and Naumann [1] proposed the model program syntax and semantics in JML for verification of higher order methods. Their idea for reasoning about HOMs is adapted from Büchi and Weck's grey-box approach [3] , [4] , [5], which was the original work about the specification and verification of such HOMs. In the work of Shaner et al., the verification of HOM is prescribed as two tasks, the first task is to verify the HOM implements the model program specification and the second task is to copy rule which copies the model program to call site. It also uses copy rule to verify the calls to HOMs at client side. In the copy rule the body of the model program is substituted for the HOM call at client with appropriate substitutions. But they did not implemented their idea of verification of calls to HOMs, which is done by this thesis.

Outline of Thesis

The next chapter gives details about the background of JML and model program. Chapter Three discuss about the parsing, verification of HOM with model program specification and testing of model program implementation in OpenJML. Discussion about the future work is discussed in Chapter Four and the conclusion is presented in Chapter Five.

CHAPTER 2: BACKGROUND ON JML AND MODEL PROGRAMS

Background on JML

JML is a formal behavioral interface specification language for Java, which is based on design by contract (DBC). Before calling a method the client has to guarantee certain conditions and once the method is called, the class in return assures certain properties. These contracts are achieved by specifying the method with preconditions and its postconditions.

These JML specifications are written in comment style. Special JML comments start with at-signs (`@`). If they have single line specification then it is written by starting `//@` and if the specifications are of multiple lines, then they are enclosed by `/*@ ...@*/`. As the specifications are written as comments, so these are ignored by a Java compiler whereas a JML compiler understands them and checks the behavior of the methods against these specifications, and if the behavior of the method does not align with these specifications, then it throws an error. Sample code with single and multiple lines of JML specifications are shown in Figure 2.1

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

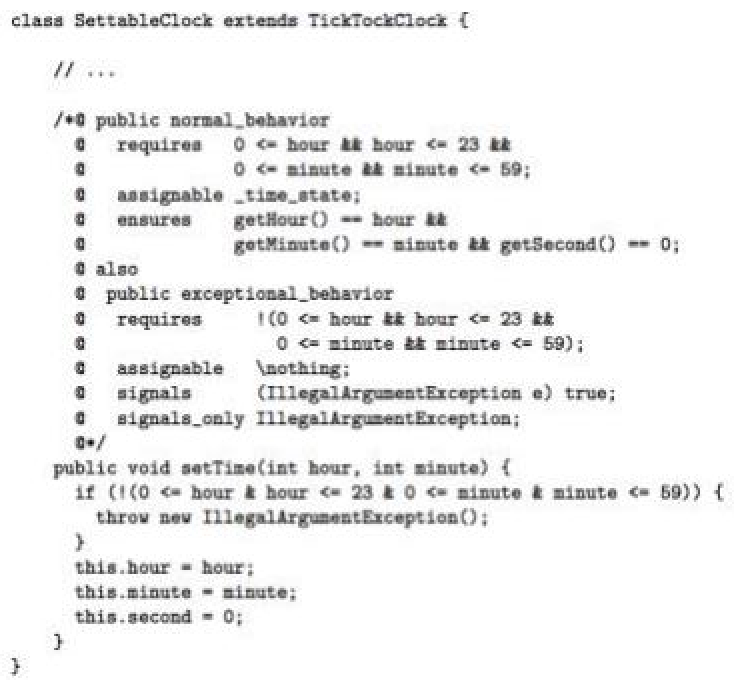
Figure 2.1: Pre- and Postconditions of `sqrt()` method (quoted from [12])

As shown in Figure 2.1, the precondtion is `x>=0.0`, which means that before calling the `sqrt()` method the argument, `x`, must be non-negative. So any client calling the `sqrt()` method has to satisfy this preconditions. In return the `sqrt()` method ensures that its postconditions are true after the end of the method. These postconditions are further classified into normal postconditions and exceptional postconditions. Normal postconditions ensure that they are true when method executes and terminates normally without any exception being thrown. The normal postcondition of `sqrt()` method is shown in figure 2.2, where these are true when no exception occurs.

```
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
```

Figure 2.2: Normal postconditions of `sqrt()` method (quoted from [12])

The second category of postconditions are exceptional postconditions, which are specified by using the `signals` clause. In general Java allows a class or a method to throw run time exceptions but JML only allows a method to throw runtime exceptions only when it is either specified in a method `throws` clause or specified in a `signals_only` clause. By default the `signals_only` clause allows the exceptions that are mentioned in method's `throws` clause. As shown in figure 2.1, the `sqrt()` method doesn't have any throw clause so it doesn't allow any exception to be thrown. An example of an exceptional postconditions is shown in figure 2.3, where it is shown that the specifications are `true` when the method throws an `IllegalArgument` exception. This prior knowledge on two categories of postconditions helps an user in writing the postconditions in the model program specifications and in the refining statements.



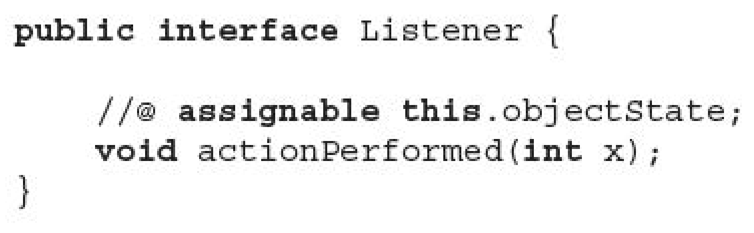
```
class SettableClock extends TickTockClock {

    // ...

    /*@ public normal_behavior
      @   requires    0 <= hour && hour <= 23 &&
      @               0 <= minute && minute <= 59;
      @   assignable _time_state;
      @   ensures     getHour() == hour &&
      @               getMinute() == minute && getSecond() == 0;
      @ also
      @  public exceptional_behavior
      @   requires      !(0 <= hour && hour <= 23 &&
      @                   0 <= minute && minute <= 59);
      @   assignable   \nothing;
      @   signals      (IllegalArgumentException e) true;
      @   signals_only IllegalArgumentException;
      @*/
    public void setTime(int hour, int minute) {
      if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
        throw new IllegalArgumentException();
      }
      this.hour = hour;
      this.minute = minute;
      this.second = 0;
    }
}
```

Figure 2.3: Exceptional postconditions example (quoted from [13])

Model Programs

Model programs are special types of method specifications which are designed to verify calls to higher order methods. HOMs are methods that makes calls to mandatory methods. Mandatory methods are ones that are either dynamically dispatched when called or that have weak specification. A dynamically dispatched method is one that is always called using dynamic dispatch; examples are abstract methods and methods defined in an interface.

An example of a method with a weak specification is shown in figure 2.4, where the `Listener`'s `actionPerformed()` method only has an `assignable` clause, which specifies that the datagroup that belongs to `objectState` can be assigned. But it does not describes the pre- and postconditions of the methods, which makes it difficult to reason about its behavior. Such methods are known as mandatory methods and the ones which makes calls to such methods will be known as HOMs. In order to reason about calls to HOMs, a model program specifications are used. The idea of the model program specifications are adapted from Shaner et al's work[1].



```
public interface Listener {

    //@ assignable this.objectState;
    void actionPerformed(int x);
}
```

Figure 2.4: An example of a method with a weak specifications (quoted from [1])

Model Programs are method specifications that expose an abstract version of HOM's code to clients. But a model program does not expose the complete code of the HOM, it exposes only about the information of the calls to mandatory methods while hiding rest of the code implementation. Model programs are introduced into a JML specifications with the keyword `model_program`. An example of the model program is shown in Figure 2.5, where the body of the model program starts with the specification statements that start with the keyword `normal_behavior`. The specification statement's `requires` clause describes its preconditions, the `ensures` clause describes its postconditions and the `assignable` clause describes the set of datagroups that are permitted to be assigned. By using specification statements the implementation details of the code are hidden and rest of the body exposes only the code that calls the mandatory methods as shown in figure 2.5.

```
/*@ public model_program {
  @
  @  normal_behavior
  @    assignable this.count;
  @    ensures this.count == \old(this.count+1);
  @
  @  if (this.lstnr != null) {
  @    this.lstnr.actionPerformed(this.count);
  @  }
  @ }
  @*/
public void bump();
```

Figure 2.5: Model program (quoted from [1])

CHAPTER 3: SOLUTION APPROACH

This chapter details the implementation of refining statements in OpenJML and also the verification of calls to HOMs by matching the code against model program specification. The First section gives details about how refining statements are parsed in OpenJML. The second section gives details about matching the code against model program specifications. The third section gives details about testing different scenarios to check that the implementation of refining statements and the first step in verification of calls to HOMs is done properly.

Parsing

In a compiler construction, initially the code is scanned by a lexical analyzer and it is converted into meaningful tokens. These tokens are analyzed against a context-free grammar (set of production rules in a formal language) and then a syntax tree is constructed. This process is known as "parsing" and a program which does parsing is known as a "parser". In OpenJML, `JMLParser.java` is the file which parses both Java and JML specifications. In order to parse the refining statements in HOMs, first the `parseStatement()` method in `JMLParser.java` is used, because the refining statements are kinds of statements. So in the `parseStatement()` method, when the current token (obtained by using the `jmlTokenKind()` method) is equal to "refining", then the refining statements abstract tree is constructed, as shown in Figure 3.1.

In order to form a refining statement abstract tree, first obtain its access modifiers (which specify the scope of the object) of the current method using the `modifiersOpt()` method; these are stored in the field named `mods`. Then the `mods` are passed as an argument to the `parseMethod-Specs()` method to obtain the method specifications. Once the specifications are obtained, they are stored in the field named `specs`. After obtaining the specifications, each a case in the specification is checked if they has correct structure or not by using the `isNone()` method. Then the immediate statement followed by the specifications is parsed using the `parseStatement()` method and stored in the field named `stmt`.

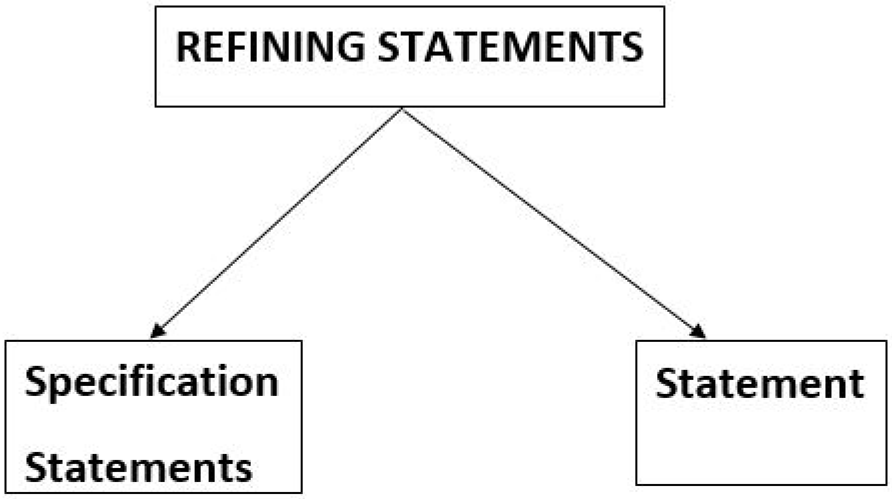


Figure 3.1: Refining Statements Semantic Tree Representation

Then construct the refining statement tree using `JMLTree`'s method called `JmlRefiningStatement()` method, as shown in the figure 3.3. This method creates an object of the `JmlRefiningStatement()` class. The fields in the `JmlRefiningStatement` class are `specs`, which contains the specification statements in the refining statements, `stmt`, which holds the statement, `jt`, which holds the jml token (which is **refining**), and `refiningSpecs`, which is used to hold the current context during walking through the parse tree for type checking (see the Type Checking section). The field `refiningSpecs` is made static in order to get the context without creating the object for the `JmlRefiningStatement` class. `JmlRefiningStatement`'s `accept()` method is used for traversals in the Visitor pattern. For traversing through refining statement we use `JMLTreeScanner`'s `visitJmlRefiningStatement()` method, as shown in figure 3.4.

Once the refining statement tree is created, it is attached to JML parse tree using `jmlF` at beginning position of current token using `pos` (which in turn calls the `pos()` method). The source code for parsing the refining statements are shown in figure 3.2

In order to print refining statements, `JMLPretty`'s `visitJmlRefiningStatement()` method which is shown in figure 3.5 is used. In this method the specifications of the refining statement are printed by starting with the JML annotation comment style `/*@` and then print the statement of the refining statement and then end with the `@*/`.

Verification of HOM implementation with model program specifications

After parsing the refining statements, the next step is to verify if the HOM's implementation satisfies the model program specification. To verify that, the first step is matching which is done in two parts. In the first part, specifications in the model program are matched with the specifications in the refining statements. In the second part, the model program's code that calls the mandatory method is matched with the code that calls the same mandatory method in the body of the HOM. This is done through `JMLAttr`'s `visitJmlModelProgramStatement()` method by traversing through a parse tree. When the tree traversal enters the model program statement, the instance of refining statements of the current method are obtained by using the `JmlRefiningStatement`'s `getRefiningStatements()` method. Once the refining statement are obtained, traverse through the specifications of the refining statements and compare it with the model program specifications. If they do not match then an error is thrown stating that the specs statements in the refining and the model program do not match each other, if they match, then the HOM have implemented the specifications in the model program. Once the specifications of the model program and the refining statements are matched, then the code that calls to a mandatory method in the model program is matched with the code that calls the same mandatory method in the body of the HOM by using `JMLAttr`'s `visitMethodDef()` method which is shown in figure 3.7.

Testing

Once parsing and the verification of HOM are done, the next step is testing the code with different scenarios. For testing `JUnit` is used. For testing `TCBase`'s `helpTC` method (shown in Figure 3.8) is used. The different test scenarios are as follows:

The first test case is shown in Figure 3.9, in which `Counter`'s `bump()` method is an example of HOM, as it calls the `actionPerformed()` method which has weak specifications(no pre- and post conditions). To reason about the verification of calls to the `bump()` method, the `bump()`'s code is checked to know if the code has the form specified by the model program. In our example, `bump()`'s code matches the model program. This is because the **refining** statement in the code matches the specification statement in the model program, and the call to `actionPerformed()` in the code matches the same mandatory call in the model program. Thus each piece of the code matches a corresponding piece of the model program. The second test case is similar to first test case except that the second test case has two HOMs namely `bump()` and `bump2()`.

For the third test case as shown in Figure 3.12, different specifications for the model program and the refining statements are written. The program throws an error stating that specs in the model program and the refining statements do not match, which is mentioned by the user. As both the errors matches with user defined errors, the test case is passed. For the fourth test case as shown

```
@Override
public JCStatement parseStatement() {
    JCStatement st;
    String reason = null;
    JmlTokenKind jtoken = jmlTokenKind();
    if (token.kind == CUSTOM) {
        boolean needSemi = true;
        if (jtoken != JmlTokenKind.ENDJMLCOMMENT) {
            int pos = pos();
            JmlSpecificationCase spc;
            if (jtoken != null)
                reason = jtoken.internedName() + " statement";
        }
        ........
        ........
        ........
        }else if (jtoken == JmlTokenKind.REFINING) {
                nextToken();
                JCModifiers mods = modifiersOpt();
                JmlMethodSpecs specs = parseMethodSpecs(mods);
                for (JmlSpecificationCase c : specs.cases) {
                    if (!isNone(c.modifiers)) {
                        jmlerror(c.modifiers.getStartPosition(),
                                getEndPos(c.modifiers),
                                "jml.no.mods.in.refining");
                    }
                }
                JCStatement stmt =  parseStatement();
                st = jmlF.at(pos).JmlRefiningStatement(REFINING,specs,stmt
                    );
                storeEnd(st, getEndPos(specs));
                needSemi = false;
```

Figure 3.2: Source code for refining statements and model programs in JMLParser.java

in Figure 3.13 the code that calls dynamically-dispatched method by HOM is missed in model program, which causes an error that is mentioned by the user.

```
@Override
public JmlRefiningStatement JmlRefiningStatement(JmlTokenKind jt,
    JmlMethodSpecs specs, JCStatement stmt) {
    return new JmlRefiningStatement(pos,jt,specs,stmt);
}

/** This class represents Refining statement
 */
public static class JmlRefiningStatement extends JmlAbstractStatement {
    public JmlMethodSpecs specs;
    public JCStatement stmt;
    public JmlTokenKind jt;
    public static JmlRefiningStatement refiningSpecs;

    protected JmlRefiningStatement(int pos,JmlTokenKind jt ,JmlMethodSpecs
         specs, JCStatement stmt) {
        this.pos = pos;
        this.specs = specs;
        this.stmt = stmt;
        this.jt = jt;
    }

    public static JmlRefiningStatement getRefiningStatements(){
        return refiningSpecs;
    }

    @Override
    public void accept(Visitor v) {
        if (v instanceof IJmlVisitor) {
            ((IJmlVisitor)v).visitJmlRefiningStatement(this);
            refiningSpecs = this;
        }

    }

    @Override
    public <R,D> R accept(TreeVisitor<R,D> v, D d) {
        return specs.accept(v,d);
    }

}
```

Figure 3.3: Source code for refining statements in JMLTree.java

```
//Refining block
public void visitJmlRefiningStatement(JmlRefiningStatement that) {
    scan(that.specs);
    scan(that.stmt);
}
```

Figure 3.4: Source code for traversing the refining statements tree in JmlTreeScanner.java

```
//Refining statement
public void visitJmlRefiningStatement(JmlRefiningStatement that) {
    try{
       print("/*@\n\t");
       print(that.jt.toString().toLowerCase()+" ");
       JmlSpecificationCase spcCase = null;
       List<JmlSpecificationCase> spcCases = that.specs.cases;
       for(int i=0;i<spcCases.length();i++) {
           spcCase = spcCases.get(i);
            if(i==0)
                print(spcCase+"\n");
            else
                print("\t"+spcCase+"\n");
        }
       print("\t*/\n");
        that.stmt.accept(this);
    } catch (IOException e) { perr(that,e); }
}
```

Figure 3.5: Source code for printing the refining statements in JmlPretty.java

```
public void visitJmlModelProgramStatement(JmlModelProgramStatement that) {

       JmlSpecificationCase modelProgramTree = (JmlSpecificationCase) that.
           getItem();
       List<JmlMethodClause> modelProgramclauses = modelProgramTree.clauses;


         JmlRefiningStatement refiningTree = JmlRefiningStatement.
             getRefiningStatements();
           for (List<? extends JmlSpecificationCase> l = refiningTree.specs.
               cases; l.nonEmpty(); l = l.tail){
                List<JmlMethodClause> refiningClauses = l.head.clauses;
                int i=0;
                for(JmlMethodClause spcCase : refiningClauses){
                     if(spcCase.toString().compareToIgnoreCase(
                         modelProgramclauses.get(i).toString())==0){
                         i++;
                     }else{
                          log.error(that.pos,"jml.mismatch.arguments.in.refining.
                              modelprogram");
                    }
                }

           }
     }
```

Figure 3.6: Source code for matching the specifications of the model program and the refining statements in JMLAttr.java

```
@Override
public void visitMethodDef(JCMethodDecl m) {
    .......
    .......
    .......

    if(jmethod.cases!=null){
        if(jmethod.cases.cases.head.token == JmlTokenKind.MODEL_PROGRAM){
            List<JCStatement> methodBodyStatements =  jmethod.body.stats.tail
                ;
            List<JCStatement> modelBodyStatements = jmethod.cases.cases.head.
                block.stats.tail;
            if(methodBodyStatements.size()==modelBodyStatements.size()){
                for(JCStatement methodBodyStatement : methodBodyStatements){
                    for(JCStatement modelBodyStatement : modelBodyStatements)
                        {
                        if(methodBodyStatement.toString().compareTo(
                            modelBodyStatement.toString())!=0){
                            log.error(m.pos, "jml.mismatch.refining");
                        }
                    }
                }
            }else{
                log.error(m.pos, "jml.mismatch.refining");
            }
        }
    }
    .......
    .......
    .......
}
```

Figure 3.7: Source code for matching the code that calls mandatory method in the model program and in the body of HOMs in JMLAttr.java

```
package org.jmlspecs.openjmltest.testcases;

import org.junit.Test;

import com.sun.org.apache.xpath.internal.axes.WalkerFactory;
import com.sun.tools.javac.comp.JmlAttr;
import com.sun.tools.javac.comp.JmlEnter;
import com.sun.tools.javac.tree.JCTree;
import com.sun.tools.javac.util.List;
import com.sun.tools.javac.util.Log;

import javax.tools.Diagnostic;
import javax.tools.JavaFileObject;

import org.jmlspecs.openjml.*;
import org.jmlspecs.openjmltest.TCBase;
import org.jmlspecs.openjmltest.TestJavaFileObject;
import org.jmlspecs.openjmltest.ParseBase;

import com.sun.tools.javac.parser.JmlScanner;
import com.sun.tools.javac.parser.Parser;
import static org.junit.Assert.fail;

public class TestRefiningStatements extends TCBase{

  @Override
  public void setUp() throws Exception {
      super.setUp();
      print = false;
  }
```

Figure 3.8: Source code for testing - part a

```
@Test
public void testModelProgramWithOneMethod(){
        helpTC(" class Counter {  \n"
        +"      private /*@ spec_public @*/  int count = 0;  \n"
        + "     private /*@ spec_public @*/ Listener lstnr = null;  \n"
        + "     /*@ assignable this.lstnr;  \n"
        + "       @ ensures this.lstnr == lnr;  \n"
        + "      @*/  \n"
        + "     public void register(Listener lnr){  \n"
        + "             this.lstnr = lnr;  \n"
        + "     }  \n"
        +"      /*@ public model_program {  \n"
        +"              @ normal_behavior  \n"
        +"              @ assignable this.count;  \n"
        +"              @ ensures this.count == \\old(this.count+1);  \
           n"
        + "             @ if (this.lstnr==null)  {  \n"
        + "                     @ this.lstnr.actionPerformed(this.
           count);  \n"
        + "             @ }\n"
        +"      @ }  \n"
        +"      @*/  \n"
        +"      public void bump(){\n"
        +"              /*@ refining normal_behavior  \n"
        +"              @ assignable this.count;  \n"
        +"              @ ensures this.count == \\old(this.count+1);  \
           n"
        +"              @*/  \n"
        + "             this.count  = this.count+1;  \n"
        + "             if (this.lstnr==null)  {  \n"
        + "                     this.lstnr.actionPerformed(this.count)
           ;  \n"
        + "             }\n"
        +"      }  \n"
        +"} \n"
        + "interface Listener{  \n"
        + " void actionPerformed(int x);  \n"
        + "} \n"
        );
}
```

Figure 3.9: Source code for testing - part b

```
@Test
public void testModelProgramWithTwoMethods(){
        helpTC(" class Counter { \n"
        +"      private /*@ spec_public @*/  int count = 0; \n"
        + "     private /*@ spec_public @*/ Listener lstnr = null; \n"
        + "     /*@ assignable this.lstnr; \n"
        + "       @ ensures this.lstnr == lnr; \n"
        + "      @*/ \n"
        + "     public void register(Listener lnr){ \n"
        + "             this.lstnr = lnr; \n"
        + "     } \n"
        +"      /*@ public model_program { \n"
        +"              @ normal_behavior \n"
        +"              @ assignable this.count; \n"
        +"              @ ensures this.count == \\old(this.count+1); \
           n"
        + "             @ if (this.lstnr==null) { \n"
        + "                     @ this.lstnr.actionPerformed(this.
           count); \n"
        + "             @ }\n"
        +"      @ } \n"
        +"      @*/ \n"
        +"      public void bump(){\n"
        +"              /*@ refining normal_behavior \n"
        +"              @ assignable this.count; \n"
        +"              @ ensures this.count == \\old(this.count+1); \
           n"
        +"              @*/ \n"
        + "             this.count  = this.count+1; \n"
        + "             if (this.lstnr==null) { \n"
        + "                     this.lstnr.actionPerformed(this.count)
           ; \n"
        + "             }\n"
        +"      } \n"
        +"      /*@ public model_program { \n"
        +"              @ normal_behavior \n"
        +"              @ assignable this.count; \n"
        +"              @ ensures this.count == \\old(this.count-1); \
           n"
        + "             @ if (this.lstnr==null) { \n"
        + "                     @ this.lstnr.actionPerformed(this.
           count); \n"
        + "             @ }\n"
        +"      @ } \n"
        +"      @*/ \n"
```

Figure 3.10: Source code for testing - part c

```
        +"      public void bump2(){\n"
        +"              /*@ refining normal_behavior \n"
        +"              @ assignable this.count; \n"
        +"              @ ensures this.count == \\old(this.count-1); \
           n"
        +"              @*/ \n"
        + "             this.count  = this.count+1; \n"
        + "             if (this.lstnr==null) { \n"
        + "                     this.lstnr.actionPerformed(this.count)
           ; \n"
        + "             }\n"
        +"      } \n"
        +"} \n"
        + "interface Listener{ \n"
        + " void actionPerformed(int x); \n"
        + "} \n"
        );
}
```

Figure 3.11: Source code for testing - part d

```
@Test
    public void testUnMatchedModelAndRefiningSpecifications(){
            helpTC(" class Counter { \n"
            +"      private /*@ spec_public @*/  int count = 0; \n"
            + "     private /*@ spec_public @*/ Listener lstnr = null; \n"
            + "     /*@ assignable this.lstnr; \n"
            + "       @ ensures this.lstnr == lnr; \n"
            + "      @*/ \n"
            + "     public void register(Listener lnr){ \n"
            + "             this.lstnr = lnr; \n"
            + "     } \n"
            +"      /*@ public model_program { \n"
            +"              @ normal_behavior \n"
            +"              @ assignable this.count; \n"
            +"              @ ensures this.count == \\old(this.count-1); \
                n"
            + "             @ if (this.lstnr==null) { \n"
            + "                     @ this.lstnr.actionPerformed(this.
                count); \n"
            + "             @ }\n"
            +"      @ } \n"
            +"      @*/ \n"
            +"      public void bump(){\n"
            +"              /*@ refining normal_behavior \n"
            +"              @ assignable this.count; \n"
            +"              @ ensures this.count == \\old(this.count+1); \
                n"
            +"              @*/ \n"
            + "             this.count  = this.count+1; \n"
            + "             if (this.lstnr==null) { \n"
            + "                     this.lstnr.actionPerformed(this.count)
                ; \n"
            + "             }\n"
            +"      } \n"
            +"} \n"
            + "interface Listener{ \n"
            + " void actionPerformed(int x); \n"
            + "} \n"
            ,"/TEST.java:11: Specification statements in Refining and
                Model Program do not match",19
            );

    }
```

Figure 3.12: Source code for testing - part e

```
    @Test
    public void testModelProgramWithMissingCodeInRefiningBlock(){
            helpTC(" class Counter { \n"
            +"      private /*@ spec_public @*/  int count = 0; \n"
            + "     private /*@ spec_public @*/ Listener lstnr = null; \n"
            + "     /*@ assignable this.lstnr; \n"
            + "       @ ensures this.lstnr == lnr; \n"
            + "      @*/ \n"
            + "     public void register(Listener lnr){ \n"
            + "             this.lstnr = lnr; \n"
            + "     } \n"
            +"      /*@ public model_program { \n"
            +"              @ normal_behavior \n"
            +"              @ assignable this.count; \n"
            +"              @ ensures this.count == \\old(this.count+1); \
               n"
            +"      @ } \n"
            +"      @*/ \n"
            +"      public void bump(){\n"
            +"              /*@ refining normal_behavior \n"
            +"              @ assignable this.count; \n"
            +"              @ ensures this.count == \\old(this.count+1); \
               n"
            +"              @*/ \n"
            + "             this.count  = this.count+1; \n"
            + "             if (this.lstnr==null) { \n"
            + "                     this.lstnr.actionPerformed(this.count)
               ; \n"
            + "             }\n"
            +"      } \n"
            +"} \n"
            + "interface Listener{ \n"
            + " void actionPerformed(int x); \n"
            + "} \n"
            ,"/TEST.java:16: Mandatory call in model program and in method
                body do not match",21
            );
    }

}
```

Figure 3.13: Source code for testing - part f

CHAPTER 4: DISCUSSION

OpenJML Control Flow

This section discusses the control flow of OpenJML, which is in the form of a flow chart shown in figure 4.1. This may be helpful for someone who wants to contribute to OpenJML's source code. The flow chart gives details about each and every method and class that is part of the JML compiler.

Future Work

The implementation of refining statements in OpenJML is the first step in the verification of calls to HOMs, which is matching the code with the model program specifications, has been achieved in this thesis. This work can be completed by achieving the second step in verification of calls to HOMs, which is checking if the body of each refining statements implements its specification.

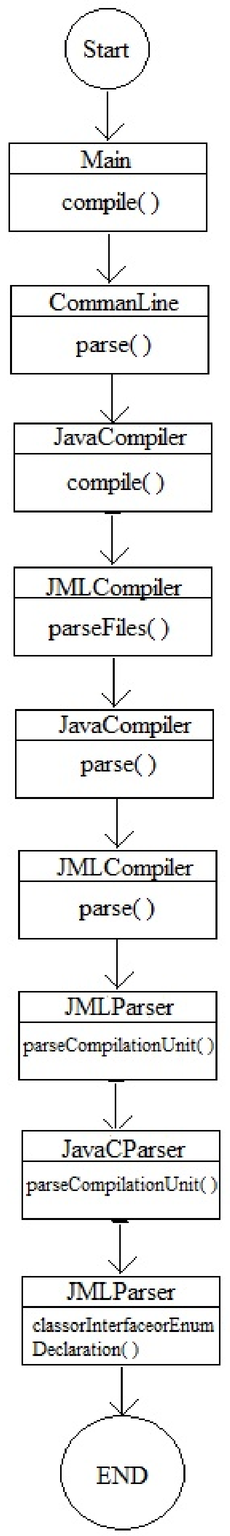


Figure 4.1: Control Flow of JML compiler construction

CHAPTER 5: CONCLUSION

The implementation of refining statements in OpenJML helps to verify HOMs with model program specifications. During reasoning about the behavior of HOMs, refining statements in the implementation help to hide the implementation's code details and expose only the specification of the code. In this thesis I have implemented the refining statements in the OpenJML tool and verified HOM implementation with model program specifications.

LIST OF REFERENCES

[1] Steve M. Shaner. *Modular verification of higher-order methods with mandatory calls specified by model programs*. Department of Computer Science, Iowa State University, January 2009.

[2] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Preliminary Design of JML: A Behavioral Interface Specification Language for Java*. ACM SIGSOFT Software Engineering Notes, 31(3):1-38, May, 2006.

[3] M. Buchi. *Safe language mechanisms for modularization and concurrency*. Technical Report TUCS Dissertations No. 28, Turku Center for Computer Science, May 2000.

[4] M. Buchi and W. Weck. *A plea for grey-box components*. Technical Report 122, Turku Center for Computer Science, Presented at the Workshop on Foundations of Component-Based Systems, Zürich, September 1997, 1997.

[5] M. Buchi and W. Weck. *The greybox approach: When blackbox specifications hide too much*. Technical Report 297, Turku Center for Computer Science, Aug. 1999.

[6] *Observer Pattern*. `https://en.wikipedia.org/wiki/Observer_pattern`.

[7] *Observer Pattern UML diagram*. `http://patterns.cs.up.ac.za/UMLdiagrams.shtml`.

[8] *Template Method Pattern UML diagram*. `http://cdn.journaldev.com/wp-content/uploads/2013/07/template-method-pattern.png`.

[9] G.W. Ernst, J. K. Navlakha, and W. F. Ogden. *Verification of programs with procedure-type parameters*. Acta Informatica,18(2):149–169, Nov. 1982.

[10] R. B. Findler and M. Felleisen. *Contracts for higher-order functions*. In Proceedings of ICFP, pages 48–59, Oct. 2002.

[11] N. Soundarajan and S. Fridella. *Incremental reasoning for object oriented systems*. In O. Owe, S. Krogdahl, and T. Lyche, editors, From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl, volume 2635 of Lecture Notes in Computer Science, pages 302–333.Springer-Verlag, 2004.

[12] Gary T. Leavens and Yoonsik Cheon. *Design by Contract with JML*. `http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf`

[13] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*. In Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures, pages 342-363. Volume 4111 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[14] C. A. R. Hoare. *An axiomatic basis for computer programming*. Communications of the ACM, 12(10):576–583, October 1969.

[15] K. R. M. Leino. *Data groups: Specifying the modification of extended state*. In OOPSLA '98 Conference Proceedings, volume 33(10) of ACM SIGPLAN Notices, pages 144–153.ACM, Oct. 1998.

[16] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[17] Guttag, John V., Horning and James J. *Larch: Languages and Tools for Formal Specification*.