Design of a JML Documentation Generator

Arun David Raghavan

TR #00-12
July 2000

Keywords: specification inheritance, refinement, browsable documentation generation, HTML, combining specifications, semantics of specifications, model-based specification, behavioral interface specification language, behavioral subtyping, formal specification languages, Eiffel, JML, Java.

1999 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, tools, JML; D.2.2 [*Software Engineering*] Design Tools and Techniques — modules and interfaces, software libraries; D.2.4 [*Software Engineering*] Software/Program Verification — formal methods, programming by contract, tools, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques;

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Contents

1 OVERVIEW **1**
- 1.1 Introduction 1
- 1.2 The Problem 2
- 1.3 Overview of the Tool 5
- 1.4 Overview of the Thesis 5

2 A SEMANTICS OF SPECIFICATION INHERITANCE AND REFINEMENT **6**
- 2.1 Introduction 6
- 2.2 Extending Specifications 6
- 2.3 JML Grammar 7
- 2.4 Semantics 7
  - 2.4.1 Desugaring Nested Specifications 9
  - 2.4.2 Desugaring Normal and Exceptional Specifications 10
  - 2.4.3 Desugaring Inheritance and Refinement 10
  - 2.4.4 Desugaring Multiple Requires Clauses 12
  - 2.4.5 Desugaring Also Combinations 14
  - 2.4.6 Combining Redundant Specifications 14
- 2.5 Specification Inheritance in JMLDoc 14
- 2.6 Summary 16

3 REQUIREMENTS FOR A DOCUMENTATION GENERATOR **17**
- 3.1 Introduction 17
- 3.2 Desugaring the Specifications 17
- 3.3 Compatibility with Javadoc Style Comments 17
- 3.4 Cross-Referenced Documentation 17
- 3.5 User Interface (Output) Design 18
- 3.6 User Options 18

4 IMPLEMENTATION OF JMLDOC **22**
- 4.1 Introduction 22
- 4.2 Overview 22
- 4.3 ClassDocWriter 22
  - 4.3.1 Creating the HTML File 22
  - 4.3.2 Generating Class Header Information 23
  - 4.3.3 Generating Member Tables 23
  - 4.3.4 Generating Field Detail 24

4.3.5 Creating Hyperlinks 24
4.4 JmlDocUnparser 25
4.5 JmlTreeSurgery 25
4.5.1 ANTLR Parse Tree 25
4.5.2 Merging Specifications 25
4.5.3 Merging "Also" Specifications 26
4.5.4 Merging "And" Specifications 31
4.5.5 Merging Conjoinable Specifications 31
4.5.6 Renaming Variables 33
4.5.7 Combining Specifications 35
4.6 Summary 35

5 RELATED WORK AND CONCLUSION **37**
5.1 Literate Programming 37
5.2 Javadoc 37
5.3 Javadoc-like Tools 38
5.4 DOC++ 38
5.5 Doclets 39
5.5.1 iDoclet 39
5.5.2 Infospheres Java Coding Standard 39
5.6 Future Work 39
5.7 Contributions 40

A REFINEMENT ORDER OF SPECIFICATION FILES **41**

B SPECIFICATION CASE DEFAULTS **42**

BIBLIOGRAPHY **43**

C ACKNOWLEDGEMENTS **45**

List of Figures

1.1 Source file Stack.java annotated with JML specifications (adapted from [12]) . 3
1.2 Specification file Stack.refines-java 4
1.3 Complete specification for the method `pop` in interface Stack 4

2.1 Top-level Method specification syntax of JML. 7
2.2 Generic specification cases. 8
2.3 Behavior specification syntax. 8
2.4 Conjoinable specification syntax. 9
2.5 Model variable declarations syntax. 9
2.6 A motivational example for nested specifications, on the left, with a desugaring on the right. 10
2.7 Desugaring nested specifications in a *normal-spec-case*. 11
2.8 Example desugaring of `normal_behavior` and `exceptional_behavior`. 12
2.9 Desugaring for extending specifications with `and`. 13
2.10 Desugaring for extending specifications with `also`. 13
2.11 Desugaring `also` combinations 15

3.1 Screen shot of a JMLDoc generated HTML file showing the class inheritance tree. 19
3.2 Sample screen shot of a JMLDoc generated HTML file. 20

4.1 Example of an AST created by ANTLR 25
4.2 Combining specifications with default clauses 27
4.3 Combined specification produced by JMLDoc 27
4.4 Example of a parse tree for inherited specification 28
4.5 Example of a parse tree for extending specification 28
4.6 Parse tree showing the merged specifications 28
4.7 Parse tree for specification of `pop` in fig. 1.1 29
4.8 Parse tree for specification of `pop` in fig. 1.2 29
4.9 Parse tree generated by `JmlTreeSurgery` for complete specification of `pop` . . . 30
4.10 Parse tree structure for inherited specification 32
4.11 Parse tree structure for extending specification 33
4.12 Parse tree structure for combined specification 34
4.13 Source file Bar.java 36
4.14 Specification file Bar.refines-java 36
4.15 Combined Specification with variable renaming 36

5.1 Example of a javadoc style documentation 38

Abstract

JML is a behavioral interface specification language designed to specify Java classes and interfaces. In JML, a Java class or interface inherits specifications from its superclass and/or superinterfaces. JML specifications for a class or module can also be refined or augmented in non-source files. As a result of this, the complete specification for a class or interface may be split over several files. To facilitate viewing of these specifications in a user-friendly manner, some documentation tools are needed to combine these specifications. This thesis gives a semantics of specification inheritance and refinement and describes an implementation of JMLDoc — a tool for generating HTML pages from the specifications. It combines specifications from inheritance and refinements and presents the complete JML specification to the user.

Chapter 1

OVERVIEW

JML [12], which stands for "Java Modeling Language," is a behavioral interface specification language (BISL)[15] designed to specify Java [2, 7] modules. JMLDoc is a tool that allows the users of these modules to view the specifications in a convenient manner, as HTML[8] documentation. This chapter discusses the need for such a tool. the motivation behind this thesis, and an overview of JMLDoc.

1.1 Introduction

JML specifications are written for Java modules. Java *modules* are compilation units containing classes and interfaces. JML specifications are used to specify the behavior of these modules. *Behavior* describes both the details of a module's interface with clients, and its behavior from the client's point of view. These specifications are a contract that a provider of a module will fulfill to the user. Figure 1.1 gives an example of a JML specification.

JML specifications are designed to be annotations in Java source files. It is also possible to have non-Java specification files that contain specifications for a Java module. Figure 1.2 shows a non-Java specification file.

We shall refer to Java classes and interfaces as *reference types* in this thesis. Syntactically, Java reference types contain fields. A *field* is a method, variable, class or an interface.

Consider the JML specification for the interface `Stack` in Figure 1.1. Comments with an immediately following at-sign, `//@`, or, C-style comments starting with `/*@`, are annotations. *Annotations*, are treated as comments by a Java compiler, but the text of an annotation, the part following the `//@` or between the annotation markers `/*@` and `@*/`, but ignoring the at-signs (`@`) at the beginning of lines between them, is meaningful to JML.

The specification for `Stack` in Figure 1.1 declares a model variable `theStack` on line 5. A *model* field is an abstraction and does not have to be implemented. However, for the purpose of specification it is treated as a Java instance field. The keyword `instance` is used to indicate that each instance of a class that implements the `Stack` interface has its own copy of the variable `theStack`. In specifications of interfaces that extend or classes that implement this interface, these model instance fields are inherited. The type of `theStack` is a `JMLObjectSequence`, which is provided in the `edu.iastate.cs.jml.models` package. Following the model declaration is an *initially* clause that specifies what must appear to be true of a field's starting value.

The specification also declares a `model instance` field called `size`. The clauses following the declaration of `size` introduce features of JML that relate model fields to concrete fields of objects. The `depends` clause states that `size` might change its value whenever `theStack` changes. The `represents` clause specifies the relation between `size` and `theStack`. The `invariant` clause states what must be true in each visible state (outside the public methods of the reference type). The `invariant` clause for the interface `Stack` states that `theStack` is non-null before and after any method invocations on a type `Stack`.

The behavioral specification for the method `pop` appears on lines 16 - 19. The keyword `public_normal_behavior` is used to implicitly state that the specification is intended for clients and that when the precondition is satisfied a call must return in the `normal` way without throwing an exception. The `requires` clause states the precondition, the `modifiable` clause states what can be modified by the method and the `ensures` clause specifies the post condition. The caller is obligated to satisfy the precondition, and the implementor is obligated to make the postcondition hold whenever the precondition holds. The specification for `pop` in Figure 1.2 states the precondition to be satisfied for an exception to be throw (exceptional behavior). In general, preconditions and postconditions in JML are written using an extended form of Java expressions. The specifications in Figure 1.2 begin with the keyword `also`. This form is intended to tell the reader that the specification given is in addition to any other specification that might have been written for the method.

1.2 The Problem

JML specifications for a Java module can be refined by JML specifications for the same module. For example, Figure 1.2 shows a refinement for the specification in Stack.java. Also, JML specifications can be inherited from a super class or a super interface. As a result of this refinement and inheritance of specifications, the complete specification for a module may not be obtained by viewing a single file or even the specification for a single module. The complete specification is obtained by combining both the refined and inherited specifications. Figure 1.3 shows the complete specification for the method `pop` in the interface Stack which is obtained by combining the specifications in Stack.java and Stack.java-refined. An user of the Java module will need to know the complete specification for that module. The user would need to look into the related files for inheritance and refinement, to get the complete specification. This is not very convenient especially if there are several specification files.

To cut down on the work of viewing several files to get the complete specification, a tool is required to automate this process. The tool should be able to collect the complete specification for a module and present it to the user.

A provider of Java modules might need to make the specifications of those modules available to clients. A suitable medium for the distribution of these specifications is the internet. The built-in classes and interfaces that are part of the Java Development Kit are documented and published on the internet using Javadoc[5] (a tool for generating documentation for Java). Javadoc requires that documentation be written in a certain way (see Section 5.2).

Existing documentation generators like Javadoc cannot interpret JML specifications. Also they do not provide the ability to combine inherited specifications with extending specifications. JML needs a documentation generator like Javadoc that can interpret JML specifications and generate documentation from the specifications. The user should be able to browse through the specifications using existing browsers. In addition to combining JML specifications, it would be to the user's benefit if the tool is able to understand Javadoc-style comments and generate documentation for those as well. This is the motivation behind providing a tool for combining JML specifications and presenting them in an appropriate format for browsing.

In summary, the goals of generating browsable HTML from JML specifications are as

```
//@ model import edu.iastate.cs.jml.models.*;

public interface Stack {

  /*@ public model instance JMLObjectSequence theStack
    @       initially: theStack != null && theStack.isEmpty();
    @*/

  public final static int MAX_SIZE = 1000;

  //@ public model instance int size;
  //@ public depends: size -> theStack;
  //@ public represents: size <- theStack.length();
  //@ public invariant: theStack != null;

  /*@ public_normal_behavior
    @   requires: !theStack.isEmpty();
    @   modifiable: size,theStack;
    @   ensures: theStack.equals(\old(theStack.trailer()));
    @*/
  public void pop( ) throws StackException;

  /*@ public_normal_behavior
    @   modifiable: size,theStack;
    @     ensures: theStack.equals(\old(theStack.insertFront(x)));
    @*/
  public void push(Object x) throws StackException;

  /*@ public_normal_behavior
    @   requires: !theStack.isEmpty();
    @   ensures: \result == theStack.first();
    @*/
   public Object top( ) throws StackException;
}
```

Figure 1.1: Source file Stack.java annotated with JML specifications (adapted from [12])

```
//@ refine: Stack <- "Stack.java";
//@ model import edu.iastate.cs.jml.models.*;
public interface Stack{

  /*@ also
    @   public_exceptional_behavior
    @     requires: theStack.isEmpty();
    @     signals: (StackException);
    @*/
  public void pop( ) throws StackException;

  /*@ also
    @   public_exceptional_behavior
    @     requires: theStack.length() == MAX_SIZE;
    @     signals: (StackException);
    @*/
  public void push(Object x ) throws StackException;

  /*@ also
    @   public_exceptional_behavior
    @     requires: theStack.isEmpty();
    @     signals: (StackException);
    @*/
  public Object top( ) throws StackException;

}
```

Figure 1.2: Specification file Stack.refines-java

```
  /*@   public_normal_behavior
    @     requires: !theStack.isEmpty();
    @     modifiable: size,theStack;
    @     ensures: theStack.equals(\old(theStack.trailer()));
    @ also
    @   public_exceptional_behavior
    @     requires: theStack.isEmpty();
    @     signals: (StackException);
    @*/
  public void pop( ) throws StackException;
```

Figure 1.3: Complete specification for the method `pop` in interface Stack

follows.

1. To automatically generate browsable HTML documentation from JML annotated source files, that can be viewed with standard browsers (e.g. on the world wide web).
2. To combine inherited and extending specifications into a complete specification that the user can view without having to look through several specification files.
3. To generate a hyperlinks that aid cross-referencing and browsing.
4. To allow the user to annotate the files, using Javadoc syntax and to generate documentation that includes it.

1.3 Overview of the Tool

This section gives an overview of the implementation of JMLDoc. In the implementation, an environment for the current compilation unit (file) is created by the JML type checker [6]. Given this environment for the current compilation unit, the tool creates a HTML document for each class, interface and inner class in the compilation unit. Each class and interface has a type attribute in the environment. The type attribute for a class stores information about its members. The JML specifications that are inherited are obtained from the type attributes of inherited members of a class or interface. These inherited specifications are combined with the current specifications and are then converted to HTML format. The combining of inherited and extending specifications is done based on a semantics that is discussed in chapter 2. Javadoc style comments are also converted into HTML. The output consists of HTML files containing documentation for each class or interface in the input compilation unit. The amount of detail in the documentation can be selected by the user. By default, documentation is generated for public and protected members and specifications. A sample HTML file generated by JMLDoc is given in Figure 3.2

1.4 Overview of the Thesis

Chapter 2 gives a semantics of refinement and inheritance of specifications in JML. This chapter is an expanded and enhanced version of an earlier technical report [14] on JML specifications. It describes a mechanism for combining JML method specifications. Chapter 3 discusses the requirements of the specification browser. Chapter 4 discusses the issues in the implementation of JMLDoc. It describes the way JMLDoc combines specifications. Chapter 5 discusses related work in this area, and compares and contrasts it to JMLDoc. It also gives the contributions of this thesis to JML and suggestions for future work in this area.

Chapter 2

A SEMANTICS OF SPECIFICATION INHERITANCE AND REFINEMENT

2.1 Introduction

This chapter describes a semantics of specification inheritance and refinement in JML, that forms the basis for combining specifications. At the end of this chapter, the steps followed by JMLDoc while combining specifications are discussed.

JML features a great deal of syntactic sugar that is designed to make specifications more expressive [11]. *Syntactic sugars* are additions to a language that make it easier for humans to use. Syntactic sugar gives the user an easier to use notation that can be easily translated (*desugared*) to produce a simpler "core" syntax. To manipulate JML specifications in tools, to understand the meaning of these sugars, and to allow the use of JML specifications in verification, this wealth of syntax must first be desugared.

The specifications that are dealt with in this chapter are the JML specifications that describe methods, i.e., method specifications. The desugaring described in this chapter does not include model programs. There is no easy way to combine them in a manner that gives the intended semantics. JML and JMLDoc also inherit and allow refinement of other declarations in classes and interfaces, but those are combined in a straightforward way (essentially a union).

2.2 Extending Specifications

A JML specification can be extended or refined by another JML specification. A subtype inherits the public and protected specifications of a supertype's public and protected members [4]. For example the file `SingleLinkedList.java` might contain some JML specifications, these specifications can be extended by `DoubleLinkList.java`. Also the specifications in `SingleLinkedList.refines-jml` can refine the specifications in `SingleLinkedList.java`.

There are two kinds of *extending-specification* for methods based on whether the specification begins with an `also` keyword or an `and` keyword. Extending specifications — augment an inherited specification from the specification of the same Java method in the superclass (if any), superinterfaces, and the refined specification.

method-specification ::= *specification* | *extending-specification*
specification ::= *spec-case-seq* [*subclassing-contract*] [*redundant-spec*]
    | *subclassing-contract* *redundant-spec*
    | *redundant-spec*
spec-case-seq ::= *spec-case* [`also` *spec-case*] ...
spec-case ::= *generic-spec-case*
    | *behavior-spec*
    | *model-program*
extending-specification ::= `also` *specification*
    | `and` *conjoinable-spec-seq* [`also` *spec-case-seq*]
    [*subclassing-contract*] [*redundant-spec*]
conjoinable-spec-seq ::= *conjoinable-spec* [`and` *conjoinable-spec*]
redundant-spec ::= *implications* [*examples*] | *examples*
implications ::= `implies_that` *spec-case-seq*
examples ::= `for_example` *example* [`also` *example*] ...

Figure 2.1: Top-level Method specification syntax of JML.

2.3 JML Grammar

The following sections give some parts of the JML grammar that are necessary to describe the semantics of inheritance and refinement of method specifications. The JML grammar is defined using an extended BNF. The following conventions are used in the grammar:

1. Nonterminal symbols are written in italics.
2. Terminal symbols appear in `teletype` font
3. Symbols within square brackets ([ ]) are optional.
4. An ellipses (...) indicates that the preceding nonterminal or group of optional text can be repeated zero or more times.

Figure 2.1 gives the top-level syntax of method specifications in JML. A method specification can be either a *specification* or an *extending specification*. A specification may consist of specification cases. A specification case may be a lightweight specification or a behavioral specification case. Figure 2.2 gives the syntax of a generic specification case. A generic specification case can be used both for lightweight specifications and in the most general form of behavior specifications. Figure 2.3 gives the syntax of a behavior specification. The conjoinable specification syntax is given in Figure 2.4. The syntax for *model-var-decls* is given in Figure 2.5.

2.4 Semantics

The following describes the overall desugaring process, at the most abstract level for a method specification. We focus first on the non-redundant parts of a specification. Section 2.4.6 discusses the redundant parts of the specifications.

Let V be a visibility level, one of `public_`, `private_`, `protected_`, or empty.

1. Desugar the use of nested *generic-spec-case-seq*, *normal-spec-case-seq*, and *exceptional-spec-case-seq* in specifications from the inside out. This eliminates nesting.

generic-spec-case ::= [*model-var-decls*] *spec-header* [*generic-spec-body*]
| [*model-var-decls*] *generic-spec-body*
spec-header ::= *requires-clause* [*requires-clause*] ...
[*when-clause*] ... [*measured-clause*] ...
| *when-clause* [*when-clause*] ... [*measured-clause*] ...
| *measured-clause* [*measured-clause*] ...
generic-spec-body ::= *simple-spec-body*
| `{|` *generic-spec-case-seq* `|}`
generic-spec-body-seq ::= *generic-spec-case* [`also` *generic-spec-case*] ...
simple-spec-body ::= *modifiable-clause-seq* *ensures-clause-seq-opt*
signals-clause-seq-opt *diverges-clause-seq-opt*
| *ensures-clause-seq* *signals-clause-seq-opt* *diverges-clause-seq-opt*
| *signals-clause-seq* *diverges-clause-seq-opt*
| *diverges-clause-seq*

Figure 2.2: Generic specification cases.

behavior-spec ::= `behavior` *generic-spec-case*
| `exceptional_behavior` *exceptional-spec-case*
| `normal_behavior` *normal-spec-case*
exceptional-spec-case ::= [*model-var-decls*] [*spec-header*]
exceptional-spec-case-body
| [*model-var-decls*] *exceptional-spec-case-body*
exceptional-spec-case-body ::= *modifiable-clause-seq*
signals-clause-seq-opt
diverges-clause-seq-opt
exceptional-spec-case-seq ::= *exceptional-spec-case*[`also` *exceptional-spec-case*] ...
normal-spec-case ::= [*model-var-decls*] [*spec-header*]
normal-spec-case-body
| [*model-var-decls*] *normal-spec-body*
normal-spec-case-body ::= *modifiable-clause-seq*
ensures-clause-seq-opt
diverges-clause-seq-opt
| *ensures-clause-seq* *diverges-clause-seq-opt*
normal-spec-case-seq ::= *normal-spec-case* [`also` *normal-spec-case*] ...

Figure 2.3: Behavior specification syntax.

```
conjoinable-spec ::= generic-conjoinable-spec
                   | behavior-conjoinable-spec
generic-conjoinable-spec ::= [ model-var-decls ] simple-spec-body
behavior-conjoinable-spec ::= behavior-kw [ model-var-decls ] simple-spec-body
                            | exceptional-behavior-kw [ model-var-decls ]
                              exceptional-simple-spec-body
                            | normal-behavior-kw [ model-var-decls ]
                              normal-simple-spec-body
exceptional-simple-spec-body ::= modifiable-clause-seq
                                 signals-clause-seq-opt
                                 diverges-clause-seq-opt
                               | signals-clause-seq diverges-clause-seq-opt
normal-simple-spec-body ::= modifiable-clause-seq
                            ensures-clause-seq-opt
                            diverges-clause-seq-opt
                          | ensures-clause-seq diverges-clause-seq-opt
```

Figure 2.4: Conjoinable specification syntax.

```
model-var-decls ::= let model-var-decl [ model-var-decl ] ...
```

Figure 2.5: Model variable declarations syntax.

2. Desugar lightweight specifications[1] to `public_behavior`. Desugar *spec-cases* beginning with V`normal_behavior` and V`exceptional_behavior` to V`behavior`. This leaves only *spec-cases* that begin with V`behavior`.

3. Combine each *extending-specification* (in a subclass or refinement) with the inherited or refined *method-specifications*, as dictated by the use of `and` or `also` in the syntax, into a single *method-specification*.

4. Desugar the use of multiple *requires-clauses* and standardize *signals-clauses*.

5. Desugar the `also` combinations within the *spec-cases*.

These steps could be run one after another, although for some purposes, such as showing specifications in a browser, one might stop after running only a few of the steps. The JMLDoc tool discussed in chapter 4 does some of the steps described below, and omits others. See chapter 4 and Section 2.5 for more details.

2.4.1 Desugaring Nested Specifications

There are several forms of specification that allow nesting. This is mainly to factor out common `requires` clauses and declarations, but the factoring also applies to `when` and `measured_by` clauses. Figure 2.6 gives a simple example.

In general, all the clauses in the top *spec-header* are copied into the each of the nested *spec-case-seqs*. The only problem is that one has to be careful to avoid variable capture. To

[1] A *lightweight* specification in JML does not use one of the V`behavior` keywords. They are designed to support incomplete specification of behavior.

```
int hailstone(int x);                       int hailstone(int x);
   requires: x > 0;                            requires: x > 0;
   {|                                          requires: x % 2 == 1;
      requires: x % 2 == 1;                    ensures: \result == 3*x + 1;
      ensures: \result == 3*x + 1;     ⇒    also
    also                                       requires: x > 0;
      requires: x % 2 == 0;                    requires: x % 2 == 0;
      ensures: \result == x / 2;               ensures: \result == x / 2;
   |}
```

Figure 2.6: A motivational example for nested specifications, on the left, with a desugaring on the right.

do this, first rename the model variables declared in the nested *spec-case-seq* so that they are not the same as any of the free variables in the outer *spec-header*.

In more detail, consider the schematic *normal-spec-case* at the top of Figure 2.7, which, assuming no renaming is needed to avoid capture, is desugared to the one at the bottom.

The same kind of desugaring applies to a *generic-spec-case* and to an *exceptional-spec-case*.

2.4.2 Desugaring Normal and Exceptional Specifications

At this point, there are no nested *spec-case*s. First, lightweight specifications are desugared by simply prefixing them with `public_behavior`. This desugars a lightweight specification to a behavior specification.

Now we desugar *spec-case*s that start with V`normal_behavior` to V`behavior` by inserting a `signals` clause of the form

```
signals: (Exception e) false;
```

which that says that no exceptions can be thrown. Similarly, we desugar *spec-case*s beginning with V`execptional_behavior` by inserting the clause "`ensures: false;`". This leaves us with non-nested generic behavior specification sequences. However, these *spec-case*s may have different visibility. So, we combine *spec-case*s with the same visibility into a single *spec-case-seq*, separating them with the `also` keyword. This leaves us with up to four *spec-case-seq*s, one for each visibility level. The general form of this desugaring is shown in Figure 2.8. The specification at the top of Figure 2.8 is desugared to the specification below.

2.4.3 Desugaring Inheritance and Refinement

Extending specifications augment an *inherited specification* from the specification of the same Java method in the superclass (if any), superinterfaces, and the refined specification. The non-redundant part of the inherited specification has up to four *spec-case*s, one for each of the four visibility levels. For each visibility level, V, the corresponding *spec-case* is formed by joining together (with `also`) all of the *spec-case-seq*s in the inherited V`behavior` *spec-case*. These *spec-case*s corresponding to the visibility levels are themselves joined together with `also`.

The rules for combining specifications from refinements are slightly different from the rules for combining specifications that are inherited from a supertype method specification.

In the case of refinements, all four visbility levels of specification are inherited by the extending specification. However, when an inherited specification comes from a supertype, only the public and protected specifications are inherited.

let $model\text{-}var\text{-}decl_{0,1}$ ... $model\text{-}var\text{-}decl_{0,v0}$
requires: $P_{0,1}$; ... requires: $P_{0,rn0}$;
when: $W_{0,1}$; ... when: $W_{0,wn0}$;
measured_by: $E_{0,1,1}$ if $MB_{0,1,1}$, ..., $E_{0,1,j_{0,1}}$ if $MB_{0,1,j_{0,1}}$; ...
measured_by: $E_{0,mbn0,1}$ if $MB_{0,mbn0,1}$, ..., $E_{0,mbn0,j_{0,mbn0}}$ if $MB_{0,mbn0,j_{0,mbn0}}$;
{|
    let $model\text{-}var\text{-}decl_{1,1}$ ... $model\text{-}var\text{-}decl_{1,v1}$
    requires: $P_{1,1}$; ... requires: $P_{1,rn1}$;
    when: $W_{1,1}$; ... when: $W_{1,wn1}$;
    measured_by: $E_{1,1,1}$ if $MB_{1,1,1}$, ..., $E_{1,1,j_{1,1}}$ if $MB_{1,1,j_{1,1}}$; ...
    measured_by: $E_{1,mbn1,1}$ if $MB_{1,mbn1,1}$, ..., $E_{1,mbn1,j_{1,mbn1}}$ if $MB_{1,mbn1,j_{1,mbn1}}$;
    $normal\text{-}spec\text{-}case\text{-}body_1$
  also ... also
    let $model\text{-}var\text{-}decl_{k,1}$ ... $model\text{-}var\text{-}decl_{k,vk}$
    requires: $P_{k,1}$; ... requires: $P_{k,rnk}$;
    when: $W_{k,1}$; ... when: $W_{k,wnk}$;
    measured_by: $E_{k,1,1}$ if $MB_{k,1,1}$, ..., $E_{k,1,j_{k,1}}$ if $MB_{k,1,j_{k,1}}$; ...
    measured_by: $E_{k,mbnk,1}$ if $MB_{k,mbnk,1}$, ..., $E_{k,mbnk,j_{k,mbnk}}$ if $MB_{k,mbnk,j_{k,mbnk}}$;
    $normal\text{-}spec\text{-}case\text{-}body_k$
|}

$\Rightarrow$

  let $model\text{-}var\text{-}decl_{0,1}$ ... $model\text{-}var\text{-}decl_{0,v0}$ $model\text{-}var\text{-}decl_{1,1}$ ... $model\text{-}var\text{-}decl_{1,v1}$
  requires: $P_{0,1}$; ... requires: $P_{0,rn0}$;
  requires: $P_{1,1}$; ... requires: $P_{1,rn1}$;
  when: $W_{0,1}$; ... when: $W_{0,wn0}$;
  when: $W_{1,1}$; ... when: $W_{1,wn1}$;
  measured_by: $E_{0,1,1}$ if $MB_{0,1,1}$, ..., $E_{0,1,j_{0,1}}$ if $MB_{0,1,j_{0,1}}$; ...
  measured_by: $E_{0,mbn0,1}$ if $MB_{0,mbn0,1}$, ..., $E_{0,mbn0,j_{0,mbn0}}$ if $MB_{0,mbn0,j_{0,mbn0}}$;
  measured_by: $E_{1,1,1}$ if $MB_{1,1,1}$, ..., $E_{1,1,j_{1,1}}$ if $MB_{1,1,j_{1,1}}$; ...
  measured_by: $E_{1,mbn1,1}$ if $MB_{1,mbn1,1}$, ..., $E_{1,mbn1,j_{1,mbn1}}$ if $MB_{1,mbn1,j_{1,mbn1}}$;
  $normal\text{-}spec\text{-}case\text{-}body_1$
also ... also
  let $model\text{-}var\text{-}decl_{0,1}$ ... $model\text{-}var\text{-}decl_{0,v0}$ $model\text{-}var\text{-}decl_{k,1}$ ... $model\text{-}var\text{-}decl_{k,vk}$
  requires: $P_{0,1}$; ... requires: $P_{0,rn0}$;
  requires: $P_{k,1}$; ... requires: $P_{k,rnk}$;
  when: $W_{0,1}$; ... when: $W_{0,wn0}$;
  when: $W_{k,1}$; ... when: $W_{k,wnk}$;
  measured_by: $E_{0,1,1}$ if $MB_{0,1,1}$, ..., $E_{0,1,j_{0,1}}$ if $MB_{0,1,j_{0,1}}$; ...
  measured_by: $E_{0,mbn0,1}$ if $MB_{0,mbn0,1}$, ..., $E_{0,mbn0,j_{0,mbn0}}$ if $MB_{0,mbn0,j_{0,mbn0}}$;
  measured_by: $E_{k,1,1}$ if $MB_{k,1,1}$, ..., $E_{k,1,j_{k,1}}$ if $MB_{k,1,j_{k,1}}$; ...
  measured_by: $E_{k,mbnk,1}$ if $MB_{k,mbnk,1}$, ..., $E_{k,mbnk,j_{k,mbnk}}$ if $MB_{k,mbnk,j_{k,mbnk}}$;
  $normal\text{-}spec\text{-}case\text{-}body_k$

Figure 2.7: Desugaring nested specifications in a *normal-spec-case*.

```
public Object top() throws EmptyException;
  public_normal_behavior
    requires: !empty();
    ensures: \result == theElems.header();
 also
  public_exceptional_behavior
    requires: empty();
    signals: (EmptyException e) true;
```

$\Rightarrow$

```
public Object top() throws EmptyException;
  public_behavior
    requires: !empty();
    ensures: \result == theElems.header();
    signals: (Exception e) false;
   also
    requires: empty();
    ensures: false;
    signals: (EmptyException e) true;
```

Figure 2.8: Example desugaring of `normal_behavior` and `exceptional_behavior`.

We desugar specifications that use the `and` syntax as shown in Figure 2.9. The same kinds of desugaring as described above is done for the *conjoinable* parts of the extending specification. A conjoinable specification may have model variable declarations and a specification case body. It is used to augment a specification. This gives us a *spec-case-seq* for each of the four visibility levels. If any of the visibility levels does not have a specification, then that *spec-case-seq* is empty. In combining the *spec-case-seq*s from the conjoinable specifications, we add each of the clauses in the conjoinable specification to the corresponding sequence of clauses in each *spec-case* of the inherited or refined *spec-case-seq*. However, *model-var-decls* from the *extending-specification* are combined, and some renaming may need to be done to avoid capture.

If there is no inherited specification for an extending or refining specification using the `and` syntax, the default specification is used as the inherited specification. This is because we can not assume that the person writing the code does not have some "implicit" specification in mind. The default specification is the lightweight specification with the default clauses.

The extending specification using the `and` syntax also allows an `also` sequence of specifications. After desugaring the `and` part, the part following the first `also` is treated as below.

The other form of extending specification starts with `also` and may have *spec-case*s with each of the four visibilities. For each visibility level, V, we combine the inherited specification with the corresponding *spec-case* of the extending specification, as shown in Figure 2.10.

2.4.4 Desugaring Multiple Requires Clauses

To make the following step clearer, we first conjoin the predicates of multiple requires clauses within each specification case.

`requires:` P_1`;` ... `requires:` P_n`;` $\Rightarrow$ `requires:` P_1 `&&` P_2 ... `&&` P_n`;`

Inherited specification:

```
public_behavior
  let model-var-decl_{1,1} ...
   model-var-decl_{1,v1}
   spec-header_1
   modifiable-clause-seq_1
   ensures-clause-seq_1
   signals-clause-seq_1
   diverges-clause-seq_1
  also ... also
  let model-var-decl_{k,1} ...
   model-var-decl_{k,vk}
   spec-header_k
   modifiable-clause-seq_k
   ensures-clause-seq_k
   signals-clause-seq_k
   diverges-clause-seq_k
```

Extending specification:

```
and
   public_behavior
   let model-var-decl-seq_B
   modifiable-clause-seq_B
   ensures-clause-seq_B
   signals-clause-seq_B
   diverges-clause-seq_B
```

$\Rightarrow$

Combined specification:

```
public_behavior
  let model-var-decl_{1,1} ...
   model-var-decl_{1,v1}
   model-var-decl-seq_B
   spec-header_1
   modifiable-clause-seq_1
   modifiable-clause-seq_B
   ensures-clause-seq_1
   ensures-clause-seq_B
   signals-clause-seq_1
   signals-clause-seq_B
   diverges-clause-seq_1
   diverges-clause-seq_B
  also ... also
  let model-var-decl_{k,1} ...
   model-var-decl_{k,vk}
   model-var-decl-seq_B
   spec-header_k
   modifiable-clause-seq_k
   modifiable-clause-seq_B
   ensures-clause-seq_k
   ensures-clause-seq_B
   signals-clause-seq_k
   signals-clause-seq_B
   diverges-clause-seq_k
   diverges-clause-seq_B
```

Figure 2.9: Desugaring for extending specifications with `and`.

Inherited specification:

```
V behavior
   spec-case_{A_1}
  also ... also
   spec-case_{A_k}
```

Extending specification:

```
also
V behavior
   spec-case_{B_1}
  also ... also
   spec-case_{B_n}
```

$\Rightarrow$

Combined specification:

```
V behavior
   spec-case_{A_1}
  also ... also
   spec-case_{A_k}
  also
   spec-case_{B_1}
  also ... also
   spec-case_{B_n}
```

Figure 2.10: Desugaring for extending specifications with `also`.

Then we standardize each `signals` clause so that it declares an exception of type `Exception`, with a standard name. (This name must be fresh to avoid capture, but we use `e` for conciseness.) The notation $[\texttt{((}ET\texttt{)e)}/n]P$ is P with the cast expression `((`ET`)e)` substituted for free occurrences of n.

$$\texttt{signals: (}ET\ n\texttt{)}\ P\texttt{;} \quad \Rightarrow \texttt{signals: (Exception e) e instanceof } ET \texttt{ \&\& } [\texttt{((}ET\texttt{)e)}/n]P\texttt{;}$$

2.4.5 Desugaring Also Combinations

At this point, there is a single method specification, which might have one *spec-case* for each visibility level. Within each visibility level we can desugar an internal *spec-case-seq* to eliminate the use of `also` as shown in Figure 2.11. When the model variable declarations are combined, there is the possibility of variable capture. Renaming of the captured variables will have to be done.

2.4.6 Combining Redundant Specifications

Redundant specifications consist of implications and/or examples. The implications consist of *spec-case-seq*s and the examples consist of *example-seq*s.

No desugaring is needed for either the implications or the examples. The *spec-case*s and examples of the same visibility level are combined together using the `also` keyword. This gives up to four *spec-case-seq*s and *example-seq*s, one for each visibility level.

In refinements, the private example sequences and implication sequences from refinement are combined with those from the extending specification.

2.5 Specification Inheritance in JMLDoc

The documentation generator for JML combines specifications from inheritance and refinements. It follows the semantics specified above while combining specifications. However, for the sake of readability, it does not do all the steps discussed above. The following describes the steps performed by JMLDoc.

Let V be a visibility level, one of `public_`, `private_`, `protected_`, or empty.

1. Desugar *spec-case*s beginning with V`normal_behavior` or V`exceptional_behavior` to V`behavior` if they are conjoined with a `signals` or an `ensures` clause respectively.

2. Combine each *extending-specification* (in a subclass or refinement) with the inherited or refined *method-specification*s, as dictated by the use of `and` or `also` in the syntax, into a single *method-specification*.

JMLDoc converts V`normal_behavior` and V`exceptional_behavior` to V`behavior` only when necessary. The final steps involving desugaring multiple clauses are omitted by JMLDoc to make the specifications more readable. Also JMLDoc does not eliminate nested sequences, because the user may not benefit very much from this desugaring. Keeping the nested specifications makes the specification more readable.

Vbehavior
    let D_1^*
    requires: P_1 ;
    when: $W_{1,1}$; ... when: $W_{1,wn1}$;
    measured_by: $E_{1,1,1}$ if $MB_{1,1,1}$, ..., $E_{1,1,j_{1,1}}$ if $MB_{1,1,j_{1,1}}$; ...
    measured_by: $E_{1,mbn1,1}$ if $MB_{1,mbn1,1}$, ..., $E_{1,mbn1,j_{1,mbn1}}$ if $MB_{1,mbn1,j_{1,mbn1}}$;
    modifiable: $SR_{1,1,1}$ if $M_{1,1,1}$, ... $SR_{1,1,m_{1,1}}$ if $M_{1,1,m_{1,1}}$; ...
    modifiable: $SR_{1,mn1,1}$ if $M_{1,mn1,1}$, ... $SR_{1,mn1,m_{1,mn1}}$ if $M_{1,mn1,m_{1,mn1}}$;
    ensures: $Q_{1,1}$; ... ensures: $Q_{1,en1}$;
    signals: (Exception e) $EX_{1,1}$; ... signals: (Exception e) $EX_{1,sn1}$;
    diverges: $T_{1,1}$; ... diverges: $T_{1,dn1}$;
also ... also
    let D_k^*
    requires: P_k ;
    when: $W_{k,1}$; ... when: $W_{k,wnk}$;
    measured_by: $E_{k,1,1}$ if $MB_{k,1,1}$, ..., $E_{k,1,j_{k,1}}$ if $MB_{k,1,j_{k,1}}$; ...
    measured_by: $E_{k,mbnk,1}$ if $MB_{k,mbnk,1}$, ..., $E_{k,mbnk,j_{k,mbnk}}$ if $MB_{k,mbnk,j_{k,mbnk}}$;
    modifiable: $SR_{k,1,1}$ if $M_{k,1,1}$, ... $SR_{k,1,m_{k,1}}$ if $M_{k,1,m_{k,1}}$; ...
    modifiable: $SR_{k,mnk,1}$ if $M_{k,mnk,1}$, ... $SR_{k,mnk,m_{k,mnk}}$ if $M_{k,mnk,m_{k,mnk}}$;
    ensures: $Q_{k,1}$; ... ensures: $Q_{k,enk}$;
    signals: (Exception e) $EX_{k,1}$; ... signals: (Exception e) $EX_{k,snk}$;
    diverges: $T_{k,1}$; ... diverges: $T_{k,dnk}$;

$\Rightarrow$

Vbehavior
   let D_1^* ... D_k^*
   requires: P_1 || P_2 ... || P_k ;
   when: (P_1 ==> $W_{1,1}$) && ... && (P_1 ==> $W_{1,wn1}$)
        && ... && (P_k ==> $W_{k,1}$) && ... && (P_k ==> $W_{k,wnk}$) ;
   measured_by: $E_{k,1,1}$ if P_1 && $MB_{k,1,1}$, ..., $E_{k,1,j_{k,1}}$ if P_1 && $MB_{k,1,j_{k,1}}$,
        ..., $E_{k,mbnk,1}$ if P_k && $MB_{k,mbnk,1}$,
            ..., $E_{k,mbnk,j_{k,mbnk}}$ if P_k && $MB_{k,mbnk,j_{k,mbnk}}$;
   modifiable: $SR_{k,1,1}$ if P_1 && $M_{k,1,1}$, ... $SR_{k,1,m_{k,1}}$ if P_1 && $M_{k,1,m_{k,1}}$,
        ..., $SR_{k,mnk,1}$ if P_k && $M_{k,mnk,1}$,
            ... $SR_{k,mnk,m_{k,mnk}}$ if P_k && $M_{k,mnk,m_{k,mnk}}$;
   ensures: (P_1 ==> $Q_{1,1}$) && ... && (P_1 ==> $Q_{1,en1}$)
        && ... && (P_k ==> $Q_{k,1}$) && ... && (P_k ==> $Q_{k,enk}$) ;
   signals: (Exception e) (P_1 ==> $EX_{1,1}$) && ... && (P_1 ==> $EX_{1,sn1}$)
        && ... && (P_k ==> $EX_{k,1}$) && ... && (P_k ==> $EX_{k,snk}$) ;
   diverges: (P_1 ==> $T_{1,1}$) && ... && (P_1 ==> $T_{1,dn1}$)
        && ... && (P_k ==> $T_{k,1}$) && ... && (P_k ==> $T_{k,dnk}$) ;

Figure 2.11: Desugaring also combinations

2.6 Summary

This chapter described the semantics of specification inheritance and refinements in JML. This is used as a basis for desugaring and combining specifications by JMLDoc. Some of the steps that are used in desugaring are not preformed by JMLDoc. This is because the user may not require the specification to be completely desugared.

Chapter 3

REQUIREMENTS FOR A DOCUMENTATION GENERATOR

3.1 Introduction

The previous chapters have introduced JML specifications and what it means to combine these specifications. A documentation generator for JML should generate browsable specifications which can be viewed using a web browser. This chapter discusses the requirements for generating browsable specifications.

3.2 Desugaring the Specifications

The main requirement is to combine the specifications from refinement and inheritance. The documentation generator should combine the specifications from the separate files in which the refinements and classes are written into a single HTML document.

The rules in chapter 2 form the basis for combining specifications. As discussed in 2.5 only some of the desugaring rules of chapter 2 are performed by JMLDoc, to preserve the original structure of the specifications. The merging of specifications is discussed in detail in section 4.5.2.

3.3 Compatibility with Javadoc Style Comments

Javadoc is a documentation generator for Java. It is the accepted standard for documenting API's in java. The documentation tool for JML should also interprets Javadoc style comments, so that a provider of a java module can write JML specifications and Javadoc documentation and be able to see both at once.

3.4 Cross-Referenced Documentation

A Java class can contain members that are instances of classes. In order for a user of a Java module to browse the specifications in a easy way the documentation should be cross-referenced. This would allow a user of a class to see the specifications of classes that are referenced by the current class with the click of a button. HTML allows cross-referencing through hyperlinks. The documentation generator for JML should make use of hyperlinks to create a cross-referenced document.

3.5 User Interface (Output) Design

The user interface or output design has been modeled on that produced by Javadoc [5] . The HTML file for a class has the class hierarchy at the very top of the page. This allows the user to see the inheritance tree.

The inheritance tree is followed by a series of tables. Each series of tables consists of a table showing the members of the current class followed by tables showing the members inherited from classes in the inheritance hierarchy. The members in the tables are hyperlinked to provide easy access to the full documentation for those members. The tables naming the inner classes and interfaces (if any) are displayed first followed by the tables for fields, constructors and finally methods.

The tables are linked to the documentation for each member of the class/interface. The documentation for the methods is the most interesting. It consists of the Javadoc style documentation followed by the complete JML specification for that method if it exists. Figures 3.1 and 3.2 show a sample HTML file generated from HTML specifications.

3.6 User Options

The tool is used to generate browsable documentation. This means creating a HTML file for each reference type. Sometimes more than one reference type can appear in a single file. The tool creates HTML files for each reference type.

A reference type has public, private, protected and package protected members. A JML specification can also be tagged as public, private, protected or package protected. The documentation tool produces documentation for the public and protected members by default. It also generates the complete specification for public and protected parts of the JML specifications. A provider of a package might want to document the private members and private specifications. To allow the user increased flexibility in generating documentation various options are provided by the tool.

An option is given to the user to allow documentation for inherited members (members that are not overridden) to be generated. By default the detailed documentation for these members is not printed. A table below the description table in the output indicates which members are inherited from which superclass and/or superinterfaces. The user can choose to suppress the documentation for the private and protected members and specifications.

The following gives the options available and the result of using each option.

Command-line Option	Action
`private`	This tells JMLDoc to generate browsable documentation for the private members of a class or interface.
`noprot`	This option prevents the browsable documentation for protected members from being generated.
`nopkgprot`	This option prevents the browsable documentation for package protected members from being generated.
`inherited`	This causes JMLDoc to generate browsable documentation

Class edu.iastate.cs.jml.models.JMLChar

```
java.lang.Object
|
+--edu.iastate.cs.jml.models.JMLChar
```

extends java.lang.Object
implements edu.iastate.cs.jml.models.JMLType

a reflection of Char that implements JMLType. By Curtis Clifton with extensive input from the JML Seminar participants at Iowa State University, June–July 1999.

Fields Index	
`char`	**`theChar`**
`char`	**`value`**

Fields inherited from java.io.Serializable
`serialVersionUID`

Constructor Summary
`JMLChar ( java.lang.Character inChar )`
`JMLChar ( int inInt )`
`JMLChar ( char c )`
`JMLChar ( )`

Figure 3.1: Screen shot of a JMLDoc generated HTML file showing the class inheritance tree.

Interface edu.iastate.cs.jml.models.JMLObjectType

extends edu.iastate.cs.jml.models.JMLType

Objects that are containers of object references. It is the intention that classes that implement JMLObjectType be "containers of objects", in the sense that the user is only interested in the "object references", (addresses) themselves as the elements in the container. (This is in opposition to the intention of classes that implement JMLValueType.) With object containers, the object references are copied in operations that create new instances of the container classes, e.g., clone(). So there is no "deep, || shallow, copy" with classes that implement JMLObjectSet. *

Fields inherited from java.io.Serializable
`serialVersionUID`

Method Summary	
`boolean`	`equals ( java.lang.Object ob2 )`
`java.lang.Object`	`clone ( )`

equals

public boolean equals (java.lang.Object ob2)
public_normal_behavior
ensures: \result <==> (* ob2 is not distinguishable from this, except by using mutation or == *);
also
public_normal_behavior
ensures: \result <==> (* ob2 and this contain the same object references *);
implies_that
public_normal_behavior
requires: s2 != null;
ensures: (s2.equals(this)) == \result;
also
requires: s2 == this;
ensures: \result <==> true;

clone

public java.lang.Object clone ( )
public_normal_behavior

Figure 3.2: Sample screen shot of a JMLDoc generated HTML file.

for the inherited (non-overridden) members of a class or interface.

The following gives a sample command line showing how JMLDoc would be called to create documentation for all the members of class `Stack` in file `Stack.java`.

```
jml --html --private --inherited Stack.java
```

Chapter 4

IMPLEMENTATION OF JMLDOC

4.1 Introduction

The JML specifications from refinement and inheritance can be combined into a complete specification using the semantics discussed in chapter 2. This chapter discusses the key issues in the implementation of JMLDoc.

4.2 Overview

The classes used in the JMLDoc are part of the `edu.iastate.cs.jml.checker.jmldoc` package. This package contains the `HMTLGenerator` and the `ClassDocWriter` classes. The `HMTLGenerator` is called from the `JmlParserDriver` to generate the documentation for a compilation unit. The `HTMLGenerator` creates a `ClassDocWriter` for each class/interface in the compilation unit. The `ClassDocWriter` takes the environment generated by the typecheker and creates the HTML document for the given class. The `ClassDocWriter` uses `JmlTreeSurgery` for merging specifications. `JmlTreeSurgery` combines the specifications from refinement and inheritance into a complete specification. The complete specification is then formatted in HTML using the `JmlDocUnparser` class.

4.3 ClassDocWriter

A `ClassDocWriter` instance is created for each class. The `ClassDocWriter` instance generates the documentation for a class by interpreting the contents of the environment for the given class. The `ClassDocWriter` is created by passing it the filename and environment for the compilation unit. The following sections describe various functions of the `ClassDocWriter`.

4.3.1 Creating the HTML File

More than one HTML file can be created for a given class. This is because specifications for a class can be extended or refined by specifications in more than one file. Therefore, the approach taken is to generate the HTML documentation based on the filename suffix of the compilation unit, for example, if the compilation unit is `Test.jml`, and it has a class `Test` defined in it, the documentation file `Test.jml.html` will be created. However the most refined file, that is, the specification file in a directory that is not further refined is treated as the base file for documentation. This means that if we had the specification files

`Test.jml-refined`, `Test.jml` and `Test.java`, then the corresponding `.html` files that would be created are `Test.jml-refined.html`, `Test.jml.html` and `Test.html`. The refinement order for specification files in JML is given in Appendix A. In the implementation, the class `SuffixMatcher` has methods that are used to find whether a file is the most refining file.

The method `printAllMembers` is called to generate the documentation for a reference type. This calls methods for printing the HTML header information. This header information builds the basic HTML *head* and *body*. Typically the head has some information carrying the title of the page and other information such as the HTML version, name of the program that generated the file, etc. The body of the file contains the actual HTML data that is presented to the user. Class header information (section 4.3.2) is generated by calling the method `printClassHeader`.

4.3.2 Generating Class Header Information

The `ClassDocWriter` generates header information for each class or interface. Header information consists of the class or interface name, the package it belongs to (if any), an inheritance tree (showing superclasses, if any), and interfaces that it implements or extends. The `ClassDocWriter` generates the class inheritance tree, which consists of a path following the class's superclasses up to `java.lang.Object`. An inheritance tree for the class `JMLChar` can be seen at the top of Figure 3.1. The inheritance tree for a class is generated by using information stored in the type attribute for each class. The type attribute is obtained from the environment by specifying the class *key*, which is the name of the class. All the information for printing the header is obtained from the type attribute for the class.

The information following the inheritance tree tells the user which classes are being extended by the current class. It also list the interfaces that the current class implements. Documentation for a interface lists the superinterfaces — interfaces that the current interface extends.

4.3.3 Generating Member Tables

Members of a class are methods, inner classes, constructors and variables. In addition to this, JML adds the following JML declarations: axiom pragmas, initializer specifications, and static initializer specifications. Tables are generated for member variables, constructors and methods. A table gives a line of information describing each field. The tables are like indices for the complete documentation. Clicking on a table entry takes the reader to the complete documentation for that particular entry. The method `printAllMembers` calls the methods `printInnerClassTable`, `printFieldsTable`, `printConstructorTable`, `printMethodTable` to generate documentation for the various members.

The field (variable) table has entries for each variable declared in the current class. Each entry gives the name of the variable and its type. Variables that are inherited are put in another table which is labelled by the name of the class from which they are inherited. The variables for the current class are filtered from the environment. The method `getFilteredTypeMap` is called to filter out the inherited variables. The method `addFieldsTableMember` is used to add a new variable entry into the variable table.

Following the variable tables are inner class tables (if any). Inner class tables are followed by constructor and method tables. Constructor tables only differ from method tables in that they do not have a return type. A method table has entries for each method. Each entry gives the return value of the method and the method signature. Section 4.3.5 describes how hyperlinks are created. The method `getMethodSignature` returns a hyperlinked method signature. The hyperlinks for the formal parameter types of a method are set to the HTML files for those types. A line describing each field is also printed in the table. This line comes from the first line of javadoc style documentation for that field (if any). The method `getFilteredMethodMap` of `ClassDocWriter` filters out the information about inherited methods. Clicking on a method entry in the method table takes the reader to the complete documentation for the method. Figure 3.1 shows an screen shot of a HTML file generated by JMLDoc. The variable table and the constructor table are seen. The underlined pieces of text are hyperlinks.

4.3.4 Generating Field Detail

Having generated the index information for the class, the `ClassDocWriter` then prints out the details of members of a class/interface. It also prints out jml fields like invariants. Options are provided to the user to allow him/her to control over the detail of specifications (see Section 3.6). The `ClassDocWriter` inherits methods for HTML formatting from the `JmlDocWriter` class, and uses the `JmlTreeSurgery` for combining specifications. The combined specifications are then formatted in HTML by the methods in `JmlDocUnparser`.

The details for each field are obtained from the type attribute for each field. A class has a type attribute which contains all the type attributes for fields defined in it. Printing the detail for each member, consists of first printing the javadoc style documentation followed by the JML specification for that field.

The JML fields (if any) are first printed after the method tables. Method `printAllMembers` calls the method `printJmlFields` to generate the documentation for the JML fields.

Following the JML fields, the documentation for variable fields is generated. The variable fields detail consists of the modifiers for the variable, the type of the variable and the variable name. This is followed by the javadoc documentation (if any) and then the JML specification (if any). The method `printFieldsDetail` prints the variable details.

Method field detail consists of the header which has the method signature followed by javadoc documentation (if any) and finally JML specifications (if any). The complete JML specification for a method consists of the specification in the current compilation unit combined with the inherited specifications from files that are refined by the current file, or from specifications of overridden methods. The specifications are merged by calling the `JmlTreeSurgery` method - `mergeSpecifications`. Section 4.5.2 discusses the merging of specifications.

Inner classes are treated differently. Instead of printing the detail of the inner classes, a new ClassDocWriter is created for each inner class, and this takes care of creating a new file for the inner class. This file can be referred to via hyperlinks.

4.3.5 Creating Hyperlinks

Hyperlinks allow ease of cross-referencing. Clicking on a hyperlink takes the user to the reference for the link. JMLDoc creates hyperlinks for reference types occurring in the documentation. The method `setHyperLink` in class `ClassDocWriter` is used to return a hyperlinked string. The input to this method is the name of the reference type. A search is made for the directory containing the most refined JML file corresponding to this reference type. A link is made to the .html file corresponding to the most refined JML file. This link is made relative to the current HTML file.

4.4 JmlDocUnparser

The `JmlDocUnparser` is a tree walker and is subclassed from `JmlUnparser`. `JmlDocUnparser` overrides methods in `JmlUnparser` in order to output text in HTML. The `ClassDocWriter` calls the `unparseMethodDeclaration` of JmlDocUnparser to return the method specifications in HTML. The `unparseDocCommentAST` method is used to return the HTML for the documentation comments and `jmlField` is used by `ClassDocWriter` to generate HTML for the JML fields.

The `unparseMethodDeclaration` of `JmlDocUnparser` calls the `unparseMethodDeclaration` of `JmlTreeSurgery` to merge specifications.

4.5 JmlTreeSurgery

JML specifications are stored in the environment created by the typechecker. The method specifications are stored as parse trees that ANTLR generates. `JmlTreeSurgery` can manipulate parse trees so that the parse trees for JML specifications can be combined. The following sections describe the implementation of `JmlTreeSurgery`.

4.5.1 ANTLR Parse Tree

This section describes the structure of the parse trees generated by ANTLR [1]. The JmlDocUnparser modifies the parse trees for the specifications in order to combine them. The parse trees consist of AST (Abstract Syntax Tree) nodes. Each AST node has a *token type*, some text (a string) and a list of children. The type field is used in the unparsers (tree walkers). The Abstract Syntax Tree has a child-sibling structure. Figure 4.1 shows a tree with `PLUS` as the root, `1` as the first child and `2` as the next sibling.

```
PLUS
 | -- 1
 ‘ -- 2
```

Figure 4.1: Example of an AST created by ANTLR

4.5.2 Merging Specifications

For the purposes of generating documentation, extending specifications need to combined with inherited specifications. The `mergeSpecifications` method of the `JmlTreeSurgery` does this. If there is no extending specification, then the inherited specification is the complete specification. The extending specification begins with an `also` or an `and` keyword.

Based on the `also` or `and` syntax, `mergeAlsoSyntax` or `mergeAndSyntax` methods of the `JmlTreeSurgery` are called. The merged specifications are then combined as discussed in section 4.5.7. This gives the complete specification for a method.

Some of the steps discussed in chapter 2 are not done by the `JmlTreeSurgery` while combining specifications. The following describes the steps that are not done and the reasons why they are omitted:

1. Desugaring nested specification cases: The desugaring of nested specifications cases is not done because the user may not benefit very much from this desugaring. Keeping the nesting of specification makes the specification more readable.

2. Desugaring Multiple clauses and desugaring of `also` clauses is not done to maintain the readability of the specifications. These transformations also require complex manipulations of the parse trees for the specifications.

4.5.3 Merging "Also" Specifications

Each specification can consist of three parts, each of which have to treated separately and then the three parts have to be combined into one complete specification. The parts that need to be merged are:

1. spec-case-seq: This is sequence of lightweight specifications, behavior specifications and/or model programs.

2. subclassing-contract: This consists of accessible and/or callable clauses. Merging of subclassing contracts is done only for refined specifications, and not for specifications that are inherited from a supertype.

3. redundant-specifications: This consists of implications and/or examples

To merge each part of the specification, a tree walker (`JmlTreeSurgery`) extracts each of the above parts separately as an AST. The following sections discuss the merging of the three parts.

Merging Spec-case Sequences

There are four *spec-case-seq*s for each of the four visibility levels. Each *spec-case-seq* is represented as an AST. In JML, specifications are tagged with a visibility modifier. When a specification for a method does not have a visibility modifier, it is intended for the same visibility level as the method. When an extending *spec-case-seq* is to be combined with an inherited *spec-case-seq*, the *spec-case*s for each visibility level are combined with the *spec-case-seq* for that visibility level. This gives four *spec-case-seq*s, one for each visibility level. If there are no *spec-case*s for a given visibility, then that *spec-case-seq* is empty.

Figures 4.4 and 4.5 show the parse trees that ANTLR generates for *spec-case*s. The `JmlTreeSurgery` merges the *spec-case-seq*s from the extending specifications and inherited or refined specifications by adding a new `also` AST node. Figure 4.6 shows the AST that will be generated by the `JmlTreeSurgery` for the given parse trees in Figures 4.4 and 4.5. Each *spec-case* in the tree, is a root and the children are not shown in the Figures.

As an example, Figure 4.7 shows the parse tree that ANTLR generates for the specification of the method `pop` in Figure 1.1. Figure 4.8 shows the parse tree for the specification of `pop` in Figure 1.2. The parse tree for the combined specification generated by `JmlTreeSurgery` is shown in Figure 4.9. The two *spec-case-seq*s are combined by introducing a new AST node `also`.

```
public_behavior
    requires: i > 0;
```

Inherited specification

```
and
  ensures: \result == true;
  signals: (Exception) false;
  diverges: false;
```

Extending specification

```
public_behavior
    requires: i > 0;
    when: true;
    measured_by: \not_specified;
    modifiable:  \nothing;
    ensures: true;
    signals: (Exception) true;
    diverges: false;
```

Inherited specification with defaults filled in

```
public_behavior
    requires: i > 0;
    when: true;
    measured_by: \not_specified;
    modifiable:  \nothing;
    ensures: \result == true;
    signals: (Exception) false;
    diverges: false;
```

Combined specification

Figure 4.2: Combining specifications with default clauses

```
public_behavior
    requires: i > 0;
    ensures: \result == true;
    signals: (Exception) false;
    diverges: false;
```

Figure 4.3: Combined specification produced by JMLDoc

```
also
    |--spec-case_A1
    `--spec-case_A2
```

Figure 4.4: Example of a parse tree for inherited specification

```
also
    |--spec-case_B1
    `--spec-case_B2
```

Figure 4.5: Example of a parse tree for extending specification

```
also
  |--also
  |    | -- also
  |    |     | -- spec-case_A1
  |    |     ` -- spec-case_A2
  |    ` -- spec-case_B1
  ` spec-case_B2
```

Figure 4.6: Parse tree showing the merged specifications

```
spec_case
|--public_normal_behavior
`--normal_spec_case
   |--requires
   |  `--!
   |     `--(
   |        `--.
   |           |--theStack
   |           `--isEmpty
   |--modifiable_seq
   |  `--modifiable
   |     `--,
   |        |--size
   |        `--theStack
   |--ensures_seq
   |  `--ensures
   |     `--(
   |        |--.
   |        |  |--theStack
   |        |  `--equals
   |        `--\old
   |           `--(
   |              `--.
   |                 |--theStack
   |                 `--trailer
   `--diverges_seq
```

Figure 4.7: Parse tree for specification of `pop` in fig. 1.1

```
EXT_also
`--spec_case
   |--public_exceptional_behavior
   `--exceptional_spec_case
      |--requires
      |  `--(
      |     `--.
      |        |--theStack
      |        `--isEmpty
      |--signals_seq
      |  `--signals
      |     |--(
      |     |--StackException
      |     `--)
      `--diverges_seq
```

Figure 4.8: Parse tree for specification of `pop` in fig. 1.2

```
also
|--spec_case
|  |--public_normal_behavior
|  `--normal_spec_case
|     |--requires
|     |  `--!
|     |     `--(
|     |        `--.
|     |           |--theStack
|     |           `--isEmpty
|     |--modifiable_seq
|     |  `--modifiable
|     |     `--,
|     |        |--size
|     |        `--theStack
|     |--ensures_seq
|     |  `--ensures
|     |     `--(
|     |        |--.
|     |        |  |--theStack
|     |        |  `--equals
|     |        `--\old
|     |           `--(
|     |              `--.
|     |                 |--theStack
|     |                 `--trailer
|     `--diverges_seq
`--spec_case
   |--public_exceptional_behavior
   `--exceptional_spec_case
      |--requires
      |  `--(
      |     `--.
      |        |--theStack
      |        `--isEmpty
      |--signals_seq
      |  `--signals
      |     |--(
      |     |--Exception
      |     `--)
      `--diverges_seq
```

Figure 4.9: Parse tree generated by `JmlTreeSurgery` for complete specification of `pop`

Merging Subclassing Contracts

A Subclassing contract consists of a sequence of accessible clauses and a sequence of callable clauses. Subclassing contracts from two specifications are combined only if one specification is a refinement of the other. First the accessible clauses (if any) are merged together, followed by the callable clauses. A new *accessible-clause-seq* AST node is created. The accessible clauses of the extending and the refined specification are linked by this new AST node. Similarly the callable clauses are linked together. The merged clauses are then returned to be merged into the complete specification

Merging Redundant Specifications

A Redundant specification consists of a implication clause and/or example clauses. First the implication clauses (if any) are merged together, followed by the example clauses. The implications and the examples consist of *spec-case-seq*s and merged as discussed in 4.5.3. The merged clauses are then returned to be merged into the complete specification

4.5.4 Merging "And" Specifications

An extending specification using the `and` syntax consists of all or some of, conjoinable specifications, specification case sequences, subclassing contracts and redundant specifications. The specification case sequences and redundant specification are merged as described earlier. The next section discusses the combining of conjoinable specifications with the *spec-case-seq* of the inherited specifications.

4.5.5 Merging Conjoinable Specifications

A conjoinable specification consists of a sequence of conjoinable specification cases. A conjoinable specification case is either a generic conjoinable specification or a behavior specification. The conjoinable specification is combined with the inherited *spec-case-seq* of the same visibility level. In case the inherited *spec-case-seq* is empty, then the conjoinable specification is merged with the default lightweight specification. A tree walker extracts the modifiable, ensures, signals and/or diverges clauses from a conjoinable specification. These clauses are then merged with the *spec-case-seq*s of the inherited specification.

If a conjoinable specification case consisting of a generic conjoinable specification is to be combined with a normal specification case, the normal specification case is desugared to a generic specification and combined with the conjoinable specification. This desugaring is done by the method `desugarBehaviorKeyword` of the class `JmlTreeSurgery`. Similar desugaring is done for exceptional specification cases.

Once the sequences for the inherited and extending specifications are merged, the corresponding sequences are then merged to form part of the complete specification.

In case the inherited specification is a normal behavior specification, and the extending specification has a signals clause, the normal behavior is desugared to a generic behavior specification and the signals clause is added to it. Similarly in case the inherited specification is an exceptional specification and the extending specification has an ensures clause, the exceptional behavior is desugared to a generic behavior. Figures 4.10, 4.11 and 4.12 show the merging of the AST's.

Figure 4.10 shows a part of a parse tree generated by ANTLR, for a specification consisting of a normal behavior and an exceptional behavior. Figure 4.11 shows a part of the parse tree

```
`--also
    |--spec_case
    |--public_normal_behavior
    |  `--normal_spec_case
    |     |--requires
    |     |  `-->
    |     |     |--i
    |     |     `--0
    |     |--ensures_seq
    |     |  `--ensures
    |     |     `-->
    |     |        |--\result
    |     |        `--0
    |     `--diverges_seq
    |        `--diverges
    |           `--false
    `--spec_case
       |--public_exceptional_behavior
       `--exceptional_spec_case
          |--requires
          |  `--<
          |     |--i
          |     `--0
          |--signals_seq
          |  `--signals
          |     |--(
          |     |--Exception
          |     |--ex
          |     |--)
          |     `--true
          `--diverges_seq
```

Figure 4.10: Parse tree structure for inherited specification

```
--EXT_and
  `--conjoinable_spec
     |--public_behavior
     |--ensures_seq
     |  `--ensures
     |     `-->
     |        |--\result
     |        `--i
     |--signals_seq
     |  `--signals
     |     |--(
     |     |--Exception
     |     |--e
     |     |--)
     |     `--false
     `--diverges_seq
      `--diverges
       `--false
```

Figure 4.11: Parse tree structure for extending specification

generated by ANTLR, for an extending conjoinable specification. The example shows how a two signals clauses are combined. Each signals clause is part of a *signals-clause-seq*. A new *signals-clause-seq* AST node is created by combining the signals clause of the inherited specification with that of the extending specification. The parse tree for the combined specification is shown in figure 4.12. The figure shows the new signals clause that has been created. The clauses for *modifiable-clause-seq*, *ensures-clause-seq* and *diverges-clause-seq* are combined in a similar manner.

The combined specification shows how a normal behavior specification is converted to a behavior specification with the inclusion of a signals clause. It also shows the desugaring of an exceptional behavior to a behavior specification with the inclusion of an ensures clause. The merging of the inherited specification with an extending conjoinable specification is done by calling the `merge_conjoinable_spec` method of the `JmlTreeSurgery`.

4.5.6 Renaming Variables

When merging two specifications, it is possible that the formal parameters in the inherited specification differ from those in the extending specification. The formal parameters in the inherited specification are renamed to be consistent with the parameters in the extending specification. Also when merging a conjoinable specification, the *model-var-decls* of the extending specification are moved into the specification case sequence. We must be careful to avoid variable capture in such cases. Figure 4.15 shows an example shows how renaming of parameters and *model-var-decls* are done for the specifications in figures 4.13 and 4.14. In this case the class variable `invert` is captured by the definition of the model variable in the refined specification. The complete specification generated renames this variable.

The method `renameVarInConjoinSpec` is called to rename the variables in a conjoinable specification case. The parameters to the method are the conjoinable specification case, the specification that is being augmented and a parameter indicating whether the conjoinable specification is a behavior conjoinable specification. The specification being augmented is

```
`--also
   |--spec_case
   |  |--public_behavior
   |  `--generic_spec_case
   |     |--requires
   |     |  `-->
   |     |     |--i
   |     |     `--0
   |     |--ensures_seq
   |     |  |--ensures
   |     |  |  `-->
   |     |  |     |--\result
   |     |  |     `--0
   |     |  `--ensures
   |     |     `-->
   |     |        |--\result
   |     |        `--i
   |     |--signals_seq
   |     |  `--signals
   |     |     |--(
   |     |     |--Exception
   |     |     |--e
   |     |     |--)
   |     |     `--false
   |     `--diverges_seq
   |        |--diverges
   |        |  `--false
   |        `--diverges
   |           `--false
   `--spec_case
      |--public_behavior
      `--generic_spec_case
         |--requires
         |  `--<
         |     |--i
         |     `--0
         |--ensures_seq
         |  `--ensures
         |     `-->
         |        |--\result
         |        `--i
         |--signals_seq
         |  |--signals
         |  |  |--(
         |  |  |--Exception
         |  |  |--ex
         |  |  |--)
         |  |  `--true
         |  `--signals
         |     |--(
         |     |--Exception
         |     |--e
         |     |--)
         |     `--false
         `--diverges_seq
            `--diverges
               `--false
```

Figure 4.12: Parse tree structure for combined specification

examined to get the list of variables used. This list is checked with the list of model variable declarations from the inherited specification. If there is a clash the corresponding variable in the conjoinable specification will be renamed. The renaming is done by appending a number to the variable to be renamed until there is no clash. For example, in Figure 4.15 the variable `invert` has been renamed to `invert0`.

4.5.7 Combining Specifications

Merging the specification leaves us with up to four *spec-case-seq*s for the specification case sequences, the subclassing contracts, four *spec-case-seq*s for the implications part of the redundant specifications and four *example-seq*s for the each of the visibility levels for the examples.

If the same specification is inherited from two interfaces, it is not duplicated in the output. The `addOrigDefs` method of the `ArrowType` class is used to add an inherited specification for a method. The `equals` method for this class checks to see if a specification is duplicated.

Based on the user selected options these specifications are combined to form the complete specification. The method `combineSpecifications` in `JmlTreeSurgery` is called to combine the various parts of the specifications into a complete specification. The combined specification is output in HTML form by the class `JmlDocUnparser`.

4.6 Summary

This chapter gave the implementation details of JMLDoc. It discussed the various classes that are part of JMLDoc. It described how the HTML documentation is generated for a class or interface. It gave the details of the implementation of `JmlTreeSurgery`, the class responsible for combining the specifications. The chapter also discussed the issues in combining specifications.

```
public class Bar
{
   public boolean invert = true;

   /*@ public_normal_behavior
     @ requires: x > 0 && invert;
     @ ensures: \result == -x;
     @*/
   public int foo(int x)
   {
       if( x >= 0)
           return -x;
       else return 0;
   }
}
```

Figure 4.13: Source file Bar.java

```
//@refine: Bar <- Bar.java"

public class Bar
{
   /*@  and
     @   public_normal_behavior
     @    let model boolean invert = false;
     @    ensures: invert == ( (y % 2) == 0 );
     @*/
   public int foo(int y);

}
```

Figure 4.14: Specification file Bar.refines-java

```
public_normal_behavior
  let model boolean invert0 = false;
    requires: y > 0 && invert;
    ensures: \result == -y;
    ensures: invert0 == ( (y%2) == 0 );
```

Figure 4.15: Combined Specification with variable renaming

Chapter 5

RELATED WORK AND CONCLUSION

This chapter discusses the related work in the area of documentation generators. It compares JmlDoc to existing documentation generators, gives suggestions for future work in this area and gives the contributions of this thesis. The discussion of documentation generators is specific to those programs that accept source files which are annotated and produce HTML or any other file format as output. The annotations may be pseudo-statements or descriptions written as comments in the native programming language.

5.1 Literate Programming

Literate programming [10] is an effort to produce better documentation for programs. The documentation is written along with the source code in the same file. Literate programming tools then parse the source files to extract the documentation and produce a formatted output, e.g., a Rich Text Format (RTF) file. Literate programming requires a typesetting language like TeX for formatting and a language like C for programming. The benefit of using a language like TeX is that typeset documentation involving mathematical symbols and diagrams can be used. JMLDoc provides some support for preformatted text. This comes from the HTML tags for preformatted text. Any preformatted documentation is written between a start tag (viz. `<pre>`) and an end tag (viz. `</pre>`).

Literate programming tools provide support for browsing, in the form of cross-references, table of contents or indexing. Jmldoc generates cross-references for reference types. However, Literate programming tools like CWEB [9] do not support JML specifications in the source code. They cannot combine JML specifications from inheritance and refinement in the way that JMLDoc does, to provide the complete specification for a reference type.

5.2 Javadoc

Javadoc [5] is a documentation generator for the Java Programming Language. The documentation is required to be written in a specified syntax within comments and is written in the source file. The documentation consists of one or more descriptive parts and a tagged description part. The tagged description part consists of a documentation tag followed by a description for that tag. Tags are used to name the authors, give version numbers, describe formal parameters and return types etc. Javadoc then generates HTML documentation from

```
/**
 * Creates a new ColorPoint object with the specified co-ordinates
 * and the specified color
 *
 * @param      x  the x co-ordinate of the ColorPoint
 * @param      y  the y co-ordinate of the ColorPoint
 * @param      c  the color of the point
 */
public ColorPoint(int x, int y, Color c) {
    ...
}
```

Figure 5.1: Example of a javadoc style documentation

the source files. Javadoc descriptions are written within Java comments. Figure 5.1 shows an example of a Javadoc comment.

Javadoc creates a hierarchy of files in a package and creates links between one file and the next. Since JML specifications are not applicable to javadoc, it does not generate documentation for such specifications. Also Javadoc does not recognize the specification refinement hierarchy in JML and cannot combine specifications into one complete specification. The JMLDoc tool recognizes Javadoc style documentation and generates documentation for these comments as well as generate documentation from the JML specifications.

Javadoc is quite extensible. An user can add new tags to the documentation and write a *doclet*, that can parse these new tags and generate documentation for that too. Section 5.5 discusses doclets for Javadoc. JMLDoc does not provide support for adding new tags or using doclets. Javadoc silently ignores bad tags. JMLDoc on the other hand, prints an error message when it encounters a tag that it does not recognize from the "standard doclet" of Javadoc.

5.3 Javadoc-like Tools

CcDoC [3] tool is a javadoc like documentation generator for C++ header files. It parses the header files and generates documentation. Documentation comments are written in a javadoc like syntax.

PolarDoc [13] is an automatic JAVA API documentation generator that is similar to javadoc, but adds many features to javadoc including a new layout for the HTML content.

Both CcDoC and PolarDoc do not recognize JML specifications and so they cannot generate documentation for JML specifications and combine specifications like JMLdoc does.

5.4 DOC++

DOC++ [16] is a documentation generator for C, C++ and Java. It generates both TeX and HTML output from C/C++ header and source files or Java class files. Some of the features of DOC++ are:

1. Hierarchically structured documentation
2. Automatic class graph generation

3. Cross references

4. High end formatting support including typesetting of equations

The documentation formats supported by DOC++ are: `/** ... */` and `///...` (where the documentation is given in "..."). DOC++ does not provide support for JML specifications.

5.5 Doclets

A Doclet is a program written in the JavaTM programming language that uses the doclet API to specify the content and format of the output of the Javadoc tool. By default, the Javadoc tool uses the "standard" doclet provided by Sun^{TM} to generate API documentation in HTML form.

5.5.1 iDoclet

iDoclet is an extension of the Javadoc 1.2 standard doclet that includes support for three new tags: `@pre`, `@post` and `@inv`. These tags are interpreted by iContract and converted into assertion check code that is inserted into the sourcecode. These tags are used by a Java preprocessor called iContract. This doclet does not recognize the JML specifications. iDoclet does not combine JML specifications from refinements and inheritance into a complete specification.

5.5.2 Infospheres Java Coding Standard

The Infospheres Java Coding Standard extends the set of standard tags. It includes tags for specifying preconditions, postconditions, class invariants, loop invariants among other tags. It includes custom tags such as `@precondition`, `postcondition`, `@requires`, `@ensures`, `@modifies` and `@examples`. It includes a debugging package that provides assertions, debugging messages, logging, carrying distributed context, and so forth. It does not support JML specifications and cannot combine JML refinements and inherited specifications.

5.6 Future Work

This section describes ideas to enhance the existing tool. Some of these ideas require adding new functionality to the existing tool. The tool can be extended by subclassing it.

The tool currently generates only HTML output from the documentation. It would be nice to have the tool generate the output in a variety of formats, for example in TeX format so that it is convenient to take hard copies. The cross referencing which is done in HTML could be simulated here by generating an index or table of contents.

The complete specification is obtained by combining the AST's from refinement and inheritance into a single AST. This AST can be serialized in an XML format. The XML file containing the AST can then be interpreted by using a stylesheet. This makes the documentation more extensible to different file formats.

Class diagrams can be generated from the source files. This can help the user understand the hierarchy of inheritance and refinements of the specifications. Also a better interface to the user could be provided. Currently the existing version of the tool does not support HTML frames. This could be changed to make cross referencing between different packages and classes better.

JMLDoc can be modified to support different views of the specification. An user might want to view only the normal behavior of a specification or only the specifications from refinements. Dynamic views for the specifications would be useful.

Currently there is no syntactic check to ensure that an extending or refining specification uses the `and` keyword or the `or` keyword. One of the unparsing phases can be modified to implement this check.

Finally, JML itself is still changing. Changes to the JML grammar might involve having to change the way the tool combines specifications since the tool deals with the AST generated by the parser, which depends on the grammar. Also new tags might be added by Javadoc to the existing set of documentation tags. Since JMLDoc recognizes javadoc tags, it will have to incorporate support for the new tags. The tool could be modified to allow any new tags to be used without signalling an error as it currently does. On the other hand, the existing tool can be used as a checker for the standard doclet the way it is now.

5.7 Contributions

The main contribution of this thesis is the combining of JML specifications from refinement and inheritance into a complete specification. This combining of JML specifications is not done by any existing documentation generator. The documentation generated is browsable using a standard HTML browser. Options are given to the user to control the visibility levels for the documentation. The user can choose the visibility levels for the members of the reference type as well as the JML specifications for a reference type.

Other contributions are the support for Javadoc style documentation and the support for embedded specifications in documentation comments.

Appendix A

REFINEMENT ORDER OF SPECIFICATION FILES

The refinement order of files in JML is as follows:

```
.refines-java
.refines-spec
.refines-jml
.java
.spec
.jml
.java-refined
.spec-refined
.jml-refined
.jml-refined
```

Files higher in the list are found by the checker first and can refine any of the others. For example, this means that a `.jml` file can refine a `.jml-refined` file. A `.spec` file can refine a `.jml` or a `.jml-refined` file. A `.java` file can refine one of the files below it in the refinement order and so on.

Appendix B

SPECIFICATION CASE DEFAULTS

The following are the specification case defaults quoted from [12].

When the various clauses of a *spec-case*, *conjoinable-spec*, or *example* are omitted, they have the defaults given in the table below.

	Default	
Omitted clause	lightweight	non-lightweight
requires:	`\not_specified`	`true`
when:	`\not_specified`	`true`
measured_by:	`\not_specified`	`\not_specified`
modifiable:	`\not_specified`	`\nothing`
ensures:	`\not_specified`	`true`
signals:	`(Exception) \not_specified`	`(Exception) true`
diverges:	`\not_specified`	`false`

A completely omitted specification is taken to be a lightweight specification. Thus one can read off the meaning of a completely omitted specification from the lightweight column of table.

BIBLIOGRAPHY

[1] *ANTLR Reference Manual.* http://www.antlr.org/, 2000.

[2] Ken Arnold and James Gosling. *The Java Programming Language.* The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[3] Ccdoc documentation generator. http://www.joelinoff.com/ccdoc/, 2000.

[4] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. A corrected version is Iowa State University, Dept. of Computer Science TR #95-20c.

[5] Lisa Friendly. Design of Javadoc. International Workshop on Hypermedia Design '95., 1995.

[6] Anand Ganapathy. Design and implementation of a JML type checker. Master's thesis, Iowa State University, 1999.

[7] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, 1996.

[8] HTML primer. http://www.ncsa.uiuc.edu/demoweb/html-primer.html, 2000.

[9] Donald Knuth and Levy S. *The CWEB System of Structured Documentation.* Addison-Wesley, 1993.

[10] Donald E. Knuth. *Literate Programming*, volume 27 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford University, 1992.

[11] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.

[12] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, Department of Computer Science, February 2000. See `www.cs.iastate.edu/~leavens/JML.html`.

[13] Polardoc, documentation generator for Java. http://www.ualberta.ca/ tgee/polardoc, 2000.

[14] Arun D. Raghavan and Gary T. Leavens. Desugaring jml method specifications. Technical Report TR #00-03, Department of Computer Science, Iowa State University, March 2000.

[15] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[16] Roland Wunderling and Malte Z ckler. Doc++, a documentation system for c/c++ and Java. http://www.zib.de/Visual/software/doc++/, 2000.

Appendix C

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my thanks to those who helped me with my work during the course of my Masters program. First and foremost, Dr. Gary T. Leavens for his guidance, patience and support throughout this research and my course of study at Iowa State University. His sense of humor and words of encouragement have often made my work more enjoyable. I would also like to thank my committee members: Dr. Don Pigozzi and Dr. Wallapak Tavanapong. I would additionally like to thank Dr. Les Miller, Dr. Robyn Lutz for their encouragement and support. I would also like to thank Melanie Eckhart, Claire Polking and my friends for their help, without which I wouldn't be writing this.