2008

A specification language design for the Java Modeling Language (JML) using Java 5 annotations

Kristina Boysen Taylor
Iowa State University

A specification language design for the Java Modeling Language (JML) using Java 5 annotations

by

Kristina Boysen Taylor

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Gary T. Leavens, Co-major Professor
Krishna Rajan, Co-major Professor
Hridesh Rajan

Iowa State University

Ames, Iowa

2008

UMI Number: 1453063

UMI Microform 1453063

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

DEDICATION

To my parents, who raised me in a house filled with love and computers.

To all of my friends over the years, who taught me that having fun is as important as working hard.

And finally, to my husband, Travis, who kept me going with love and patience, even when it seemed as though I was facing the impossible.

TABLE OF CONTENTS

LIST OF TABLES vi

LIST OF FIGURES vii

ABSTRACT ix

CHAPTER 1. OVERVIEW 1

1.1 Introduction 2

1.2 Goals 3

CHAPTER 2. REVIEW OF LITERATURE 4

2.1 Background 4

2.2 Related Work 5

CHAPTER 3. LANGUAGE DESIGN 7

3.1 Background 7

3.2 Three Approaches 8

3.2.1 The Single Annotation 8

3.2.2 The Parameter Annotation 8

3.2.3 The Clausal Annotation 9

3.3 Discussion 9

3.3.1 Consistency 10

3.3.2 Readability 10

3.3.3 Usability 11

3.3.4 Extensibility 11

3.3.5 Summary 12

3.4 Use Cases . . . 12
3.4.1 Specification Definition Modifiers versus Specification Clauses . . . 12
3.4.2 Model Methods . . . 13
3.4.3 Specification Cases . . . 14
CHAPTER 4. IMPLEMENTATION . . . 17
4.1 Prework . . . 17
4.1.1 Resolving Syntax Tree Issues . . . 17
4.1.2 Writing Bytecode . . . 18
4.2 Adding an Additional Compiler Pass . . . 18
4.2.1 Resolving Annotation Types . . . 18
4.2.2 Walking the Abstract Syntax Tree . . . 19
4.3 Translating the Annotations . . . 19
4.3.1 Writing the Parser . . . 19
4.3.2 Integrating the Parsed Elements . . . 21
4.4 Conclusion . . . 24
CHAPTER 5. CASE STUDIES . . . 25
5.1 Two Small Test Cases . . . 25
5.2 One Large Test Case . . . 26
CHAPTER 6. USING JML ANNOTATIONS FOR VERIFICATION OF WEB-BASED DATA ENTRY . . . 29
6.1 Abstract . . . 29
6.2 Introduction . . . 29
6.3 Significance . . . 30
6.4 Related Work . . . 31
6.5 Approach . . . 32
6.6 Results . . . 33
6.6.1 Checking Individual Properties . . . 33
6.6.2 Checking Properties on Form Submit . . . 36

6.7 Evaluation 37
6.8 Conclusion 38
CHAPTER 7. DISCUSSION AND SUMMARY 40
7.1 Discussion 40
7.1.1 Annotations as JML 40
7.1.2 Annotations as Java Syntax 42
7.2 Summary 42
APPENDIX. ANNOTATION SYNTAX 43
BIBLIOGRAPHY 53
ACKNOWLEDGMENTS 57

LIST OF TABLES

Table 7.1 JML2 vs. JML5 41

LIST OF FIGURES

Figure 1.1 JML2 Annotation Example 2

Figure 1.2 JML5 Annotation Example 2

Figure 3.1 Base Example 7

Figure 3.2 Single Annotation Example 8

Figure 3.3 Parameter Annotation Example 9

Figure 3.4 Clausal Annotation Example 9

Figure 3.5 Model and Invariant Base Example 12

Figure 3.6 Model and Invariant Clausal Example 13

Figure 3.7 Model Method Base Example 14

Figure 3.8 Model Method Clausal Example 14

Figure 3.9 Specification Case Base Example 15

Figure 3.10 Specification Case Clausal Example 15

Figure 3.11 Specification Case Clausal Example (with strings) 16

Figure 4.1 Partial Package Resolution 18

Figure 4.2 Annotation Accessor in JMethodDeclaration.java 19

Figure 4.3 Grammar for `jmlRequiresClause[]` 20

Figure 4.4 Partial JML Grammar 20

Figure 4.5 Partial JML5 Grammar 20

Figure 4.6 Grammar for `jmlRepresentsClause[]` 21

Figure 4.7 Parser Method 22

Figure 4.8 JMethodDeclaration.java 23

Figure 4.9 JmlMethodDeclaration.java 23
Figure 4.10 Visiting a `JmlMethodDeclaration` 24

Figure 5.1 MarkerAnnotations.java 25
Figure 5.2 SingleElementAnnotations.java 26
Figure 5.3 AAList.java 27
Figure 5.4 AAList Incorrect Constructor 28

Figure 6.1 Tapestry Template Example 33
Figure 6.2 Tapestry Java Example 34
Figure 6.3 Tapestry Web Form Example 34
Figure 6.4 Transformed `setMass(..)` Method 35
Figure 6.5 Tapestry Error Form 35
Figure 6.6 Multiple Property Web Form 36
Figure 6.7 Transformed `onSubmitFromNewDose(..)` Method 37
Figure 6.8 Tapestry Error Form 37

ABSTRACT

Design by contract specification languages help programmers write their intentions for a piece of code in a formal mathematical language. Most programming languages do not have built-in syntax for such specifications, so many design by contract languages place specifications in comments. The Java Modeling Language (JML) is one such specification language for Java that uses comments to specify contracts. However, starting with version 5, Java has introduced annotations, a syntactical structure to place metadata in various places in the code. This thesis proposes an initial design to writing JML contracts in the Java 5 annotation syntax and evaluates several criteria in the areas of specification languages and Java language design: whether these annotations are expressive enough to take advantage of annotation simplicity and tool support, and whether the annotation syntax is expressive enough to support handling a large specification language such as JML.

CHAPTER 1. OVERVIEW

Much of programming language theory has a basis in higher mathematics, as many students of computer science learn throughout their undergraduate career. Despite this, even writing a simple mathematical function in any programming language may cause the programmer to lose sight of the original goals of the function in the midst of assigning variables, defining classes, and optimizing the code. Future readers of the code can become even more confused about the intentions of a piece of code and misunderstand the requirements and assurances of a particular function.

This is where the concept of a *design by contract* specification language comes in, introduced by Bertrand Meyer in his book *Object-oriented Software Construction* [22; 4]. Programmers express the intentions of their code by specifying a contract, both what should be true before visiting a particular piece of code and also what that piece of code guarantees after it runs. Since these contracts are in a mathematical language, they act as executable documentation for the code, allowing programmers to verify whether their code contracts work with the code contracts of a system already in place.

The Java Modeling Language (JML) is one such language and mathematical verifier. It provides a mathematical language to write contracts, also known as assertions, specifically for Java code verification. Because of restrictions on how the Java language is structured, JML uses Java comments to write assertions throughout the body of the code. Starting with version 5 however, Java contains language constructs called *annotations* to provide language metadata alongside the Java language [11]. In light of this development, I suggested that we instead use the new Java 5 annotations as an alternate format for JML assertions for two reasons: these annotations will be more familiar to first-time users of JML than the custom JML comment syntax, and differentiating JML comments from regular Java comments requires quite a bit of preprocessing. Thus, simplifying both the language and the processing are the major motivations behind research into moving JML to this new syntax.

1.1 Introduction

A normal JML assertion on a method typically has what is known as a ”requires” and ”ensures” clause. The requires clause states what is required to be true before entrance into the method. The ensures clause states what the execution of the method will assert if the required prerequisites are true. Figure 1.1 has an example `deposit(..)` method with an attached JML assertion. It states that if `amount > 0`, then the method will assign to `balance`, and the sum of the initial value of `balance` and `amount` is equal to the final value of `balance`. Otherwise, if `amount <= 0`, then no variables change and the method signals an exception.

Now, compare the syntax of figure 1.1 to that of figure 1.2.

```
/*@    public normal_behavior
  @      requires amount > 0;
  @      assignable balance;
  @      ensures balance == \old(balance) + amount;
  @  also
  @    public exceptional_behavior
  @      requires amount <= 0;
  @      assignable \nothing;
  @      signals (Exception) amount <= 0;
  @*/
public void deposit(int amount) throws Exception;
```

Figure 1.1 JML2 Annotation Example

```
@Also({
  @SpecCase(
    header = "public normal_behavior"
    requires = "amount > 0;",
    assignable = "balance;",
    ensures = "balance == #old(balance) + amount;"),
  @SpecCase(
    header = "public exceptional_behavior"
    requires = "amount <= 0;",
    assignable = "#nothing;",
    signals = "(Exception) amount <= 0;")
})
public void deposit(int amount) throws Exception;
```

Figure 1.2 JML5 Annotation Example

Both figures express the same JML assertion. Figure 1.1 on the previous page expresses it in the current syntax, where assertions must appear both in "/*" comments and in "@" delimiters to ensure processing in JML. In contrast, figure 1.2 on the preceding page expresses it in the new Java 5 annotations instead. The syntax of `@Also` and `@SpecCase` is defined not in a separate JML parser but by separate Java annotation interfaces. This makes the annotations immediately accessible in the Java abstract syntax tree, making it possible for an integrated development environment (IDE) to assist with annotation construction. In addition, the annotations separate the assertions from the comments as well as other assertions from another specification language, since annotation types are resolved like classes and interfaces. Thus, the JML compiler need only parse the JML expressions inside of the quoted strings in the annotations.

1.2 Goals

I aim to answer two questions in this thesis:

- Is there a way to write JML Java 5 annotations such that they are consistent, readable, usable, and extensible?
- What changes should be made to Java 5 annotations, if any, to make them more useful with JML?

The structure of this thesis covers previous work, initial design of the annotations, and the application of the annotations. Chapter 2 covers the background of annotations and the related work to using specification languages in annotations. Chapter 3 explains how the final syntax evolved from initial ideas. Chapter 4 provides an overview of the techniques used to change the JML parser to handle the new annotations. Chapter 5 analyzes how the annotations work in practice while chapter 6 analyzes how the annotations work when used as a verification tool in web forms. Finally, chapter 7 discusses and summarizes the results.

CHAPTER 2. REVIEW OF LITERATURE

This chapter covers the background research leading up to the creation of Java 5 annotations and reviews other related approaches to place assertions inside these annotations.

2.1 Background

Extensible programming, the idea that the programmer extends the language to match the application, has existed for some time, most notably in Lisp. Similar to this idea is the macro found in C and C++, which allows the programmer to write shorthand definitions and processors [18]. Van Wyk notes that programmers have used macros extensively to translate a shorthand source language into the larger target language designated for compilation and testing. He also states that using macros and other extensions is part of a larger effort to implement *intentional programming* [28], a field originally developed by Simonyi to enable programmers to extend programming languages to meet the needs of a particular domain [39]. More recently, researchers have implemented translators in C# attributes, as demonstrated by Ghezzi and Monga in research on mapping dependencies among classes [10].

Java Specification Request (JSR) 175 originally introduced annotations into Java 5, to address the ”growing trend towards annotating fields, methods, and classes as having particular attributes that indicate they should be processed in special ways by development tools, deployment tools, or run-time libraries” [31]. Open source projects, such as Apache Tapestry 5 [2] and Hibernate [24], use these new annotations as an alternative to XML-based configuration files, simplifying configuration down to a single Java file instead of both a Java file and an XML file.

2.2 Related Work

Several techniques for integrating specification assertions in to programming languages currently exist. Sjögren [29] highlights the five basic language integration techniques for contracts:

- Write textual descriptions in comments
- Parse an abstract syntax tree from formally structured comments
- Add to the existing language
- Write libraries with assertion methods
- Transform code from an established specification language to checking code in the existing language

He later suggests using C# attributes to write assertions on classes and methods, allowing designers to swap the native specification language (OCL) for a different specification language, since the assertions are contained in parsable strings. eXtensible C# [25], C#AL [16], and nContract [13] all have expressive annotation contracts for C#; however, none of these address specification concerns in Java.

Both JSR 305 [32] and JSR 308 [34] discuss the idea of simple Java annotation code checking constructs, by suggesting such annotations as **`@NonNull`**, **`@Nullity`**, and **`@Pure`**. JSR 308 aims to populate more areas in the code with annotations, and JSR 305 aims to create a standard set of annotations to assist code checking tools. However, neither of these propose tools that would actually verify the code; instead they aim for an extension of standard Java annotations.

XVP [27] uses Java 5 annotations to manage assertions in a declarative form. While it improves on JML by representing assertions using reflection, it only defines two annotations (**`@Constraint`** and **`@Pure`**). The authors explain that this is to allow for any constraint language to be substituted in place of JML. However, this makes the constraints difficult to read; it is hard to find the starts and ends of separate clauses without clear string formatting.

Contract4J [37] goes further by splitting the contracts into three annotations: **`@Pre`** (preconditions), **`@Post`** (postconditions), and **`@Invar`** (invariants). However, its expression language is not as expressive as JML, as there are no mechanisms to denote **`assignable`**, **`pure`**, or **`model`** constructs.

OVal with AspectJ [36] defines many annotations similar to JML, such as **`@NonNull`** and the purity indicator **`@IsInvariant`**. It allows the user to select the expression language inside of the annotations from five different expression languages (BeanShell, Groovy, JavaScript, MVEL, and OGNL) and also comes with a translator from Enterprise JavaBeans 3.0 (EJB3) to OVal annotations, allowing users to put constraints on database objects. Again, while the annotations and expressions as a whole cover more than Contract4J, they are not as expressive as JML, leaving out full **`assignable`** support and **`model`** constructs altogether.

Modern Jass [26] goes the furthest in using Java 5 annotations, introducing 13 separate annotations and a much more complete expression language than previously listed works, based on the constraint language Jass [3]. Since Modern Jass and JML have this much in common, I have collaborated with Johannes Rieken, the creator of Modern Jass, to come up with a common set of Java 5 annotations for use in both Modern Jass and JML. Still, JML's language contains many more features than Jass or Modern Jass, so the current design contains a superset of annotations out of which other languages, such as Modern Jass, can choose a subset of features to support. The rest of the thesis explains this language design in further detail.

CHAPTER 3. LANGUAGE DESIGN

There are several ways to translate the comment-based JML syntax into Java 5 annotations. This chapter reviews the different proposed approaches and describes the final approach.

3.1 Background

Consider the example in figure 3.1 inspired by one in the JML Reference Manual [19]:

```
public abstract class IntList {

  //@ public model non_null int [] elements;

  //@ ensures \result == elements.length;
  public abstract /*@ pure @*/ int size();

  //@ signals (IndexOutOfBoundsException e)
  //@         index < 0 || index >= elements.length;
  public abstract /*@ pure @*/ int elementAt(int index)
                               throws IndexOutOfBoundsException;

}
```

Figure 3.1 Base Example

This example can translate into Java 5 annotations in several ways, from a simple annotation with only one field to a complex annotation with several fields. I consider several criteria when judging how usable these annotations are to the users and designers:

Consistency Annotations appear in several places, and they must be written the same way in each place.

Readability Annotations must appear without too much syntax around them so that readers can clearly find and easily read the predicate.

Usability Annotations must be flexible in how they are used: intuitive and straightforward for new users while being fully expressive for veteran users.

Extensibility Annotations must be able to be extended without disrupting the current syntax.

3.2 Three Approaches

Now I present three approaches to integrating JML into Java 5 annotations.

3.2.1 The Single Annotation

One extreme technique is to just put all of the JML assertion inside one grand annotation. As mentioned previously, XVP takes this approach by using the encompassing **`@Constraint`** annotation to hold the entirety of a JML expression [1]. An example is given in figure 3.2.

```
@JML("public model non_null int [] elements;")
public abstract class IntList {

  @JML("ensures #result == elements.length;")
  public abstract @JML("pure") int size();

  @JML("signals (IndexOutOfBoundsException e)"
            + "index < 0 || index >= elements.length;")
  public abstract @JML("pure") int elementAt(int index)
                               throws IndexOutOfBoundsException;

}
```

Figure 3.2 Single Annotation Example

3.2.2 The Parameter Annotation

The other extreme technique is to separate every single part of the annotation into its parts inside of the annotation. This was the original design of the JML annotations, but later analysis will show that going fully by this philosophy causes problems. The same example in the previous section is modified for this approach in figure 3.3 on the next page.

[1] To avoid confusion with JML's **`constraint`** keyword, this chapter will use **`@JML`** as a replacement for **`@Constraint`**

```
@Model(visibility = Visibility.PUBLIC, nonnull = true,
       value = "int [] elements;")
public abstract class IntList {

  @Ensures("#result == elements.length;")
  public abstract @Pure int size();

  @Signals(type = "IndexOutOfBoundsException", ident = "e",
           value = "index < 0 || index >= elements.length;")
  public abstract @Pure int elementAt(int index)
                        throws IndexOutOfBoundsException;

}
```

Figure 3.3 Parameter Annotation Example

3.2.3 The Clausal Annotation

Since the two previous techniques are opposite of each other, it makes sense to try to compromise between them. The driving design decision behind this approach is to give as much work to the current parser as possible without losing the type of the assertion. This means that every clause that can occur by itself has an annotation with that name and retains that structure throughout all expressions, even within strings. Again, the same example in previous sections is modified for this approach in figure 3.4.

```
@Model("public @NonNull int [] elements;")
public abstract class IntList {

  @Ensures("#result == elements.length;")
  public abstract @Pure int size();

  @Signals("(IndexOutOfBoundsException e)"
         + "index < 0 || index >= elements.length;")
  public abstract @Pure int elementAt(int index)
                        throws IndexOutOfBoundsException;

}
```

Figure 3.4 Clausal Annotation Example

3.3 Discussion

Now I evaluate each approach based on the criteria stated above.

3.3.1 Consistency

The single annotation approach is the most consistent of the three, since the annotation encompasses the entire grammar. This becomes slightly less consistent when using syntactic sugar like **`@Pure`**, since there are now two ways of specifying a single **`pure`** assertion on a method. The clausal approach is slightly less consistent than the single annotation approach, especially when it comes to using parameters for a **`@SpecCase`** (see figure 1.1 on page 2). For example, in one area, a **`requires`** clause might appear as

```
@Requires("amount > 0;")
```

but in the **`@SpecCase`** it would appear as

```
requires = "amount > 0;"
```

However, since this approach requires that the content inside the strings is the same and thus does not split up assertions, it is much more consistent than the parameter approach.

3.3.2 Readability

The clausal approach is the most readable of the three, since it strikes a compromise between the two extreme approaches. The single annotation approach obfuscates the assertion meaning by forcing the reader to read past **`@JML`**`(..)` to figure out the actual assertion type. The parameter approach separates every component of the predicate into different pieces in the annotation, eliminating the sentence-like structure of the JML expressions. The **`signals`** clause in the example illustrates this point well, since a phrase like

```
//@ signals (IndexOutOfBoundsException e)
//@         index < 0 || index >= elements.length);
```

translates into

```
@Signals(type = "IndexOutOfBoundsException", ident = "e",
         value = "index < 0 || index >= elements.length;")
```

This translation becomes even more complicated for modifiers that turn into their own language constructs, such as **`model`**. This strategy transforms a statement like

```
//@ public model non_null int [] elements;
```

into

```
@Model(visibility = Visibility.PUBLIC, nonnull = true,
       value = "int [] elements;")
```

It is unreasonable to create fields for all of the other modifiers that can attach to **`model`**. Even though these modifiers are known (see JML Reference Manual, Appendix B), even slight language redesigns would require changes in this annotation.

The clausal approach strikes a nice balance between the two, specifying the type of the assertion while not splitting the predicate into pieces. In this approach **`signals`** becomes

```
@Signals("(IndexOutOfBoundsException e)"
       + "index < 0 || index >= elements.length;")
```

while **`model`** turns into

```
@Model("public @NonNull int [] elements;")
```

a much shorter and more compact annotation.

3.3.3 Usability

The single annotation approach may be more usable by JML veterans, but it does not guide new users on the different elements of the annotations. The parameter approach is the most usable, as it lists in its specification the elements of the annotation, their types, and whether or not they are required. For example, the **`@Signals`** clause shows the user that while merely a type of exception is required, the user can also specify an identifier and a predicate. The clausal approach only specifies the type of the annotation and nothing about the inner syntax, so it is only slightly more usable than the single annotation approach.

3.3.4 Extensibility

The single annotation and clausal approaches are the most extensible, since most or all of the language extensions are handled by the parser. Thus, the parser continues to be responsible for the error checking, while the hand-written code only checks the type of the annotation and feeds all relevant

information back to the parser. In the parameter approach, a language change requires adding a new annotation component as well as new hand-written code to feed it to the parser, increasing the workload for the tool programmers and the possibility for errors.

3.3.5 Summary

Using these criteria, I conclude that the single annotation approach is not helpful to new users and that the parameter annotation approach is too complicated. This leaves the clausal annotation approach as the best compromise. In addition, since the clausal annotation forwards much of the work to the parser, it allows different projects, such as the Modern Jass project, to use the basic JML language while implementing their own language enhancements and expression grammars. Since I see this as the best approach, the rest of this chapter focuses on translating various JML language constructs into clausal annotations.

3.4 Use Cases

The running example in the previous sections does not cover many of the other JML use cases, so this section covers other cases, discussed while reviewing this technique, that have contributed to overall design decisions.

3.4.1 Specification Definition Modifiers versus Specification Clauses

The biggest problem has been dealing with class-level specification definition modifiers (like **`model`**) verses class-level specification clauses (like **`invariant`**). To see this difference, look at the example in figure 3.5 in the regular JML syntax:

```
public abstract class IntList {

  //@ public model non_null int [] elements;
  //@ public invariant elements.length < 10;

}
```

Figure 3.5 Model and Invariant Base Example

Now look at the original clausal proposal in figure 3.6.

```
@Model(header = "public @NonNull", value = "int [] elements;")
@Invariant(header = "public", value = "elements.length < 10;")
public abstract class IntList {}
```

Figure 3.6 Model and Invariant Clausal Example

It is a very difficult syntax to work with, specifically because of having to explicitly specify the modifiers in the header field. It would be easier to express the **model** field as

```
@Model("public @NonNull int [] elements")
```

However, using this approach consistently with **invariant** would look like

```
@Invariant("public elements.length < 10")
```

This looks strange to anyone familiar to Java since it attempts to assign a visibility attribute to an expression. However, since these are classifying two different JML constructs, it would not violate consistency to treat them differently. Thus, it would be acceptable to write the **model** field as above while writing the **invariant** specification as

```
@Invariant(header = "public", value = "elements.length < 10;")
```

since type specifications can only be modified with **public, protected, private, instance,** or **static**. While it would be easier to use an enumerated type for the modifiers, there is a possibility that any specification could use two of these modifiers together (when using **instance** or **static**). From a semantic point of view, the visibility modifiers are not of the same type as the **instance** or **static** modifier, making it unreasonable to create a single enumerated type and too complicated to make two types. Thus, the modifiers exist as simple parsed strings.

3.4.2 Model Methods

Another advantage of the previous approach is that it makes it easier to specify **model** methods. Compare the **model** method in the original syntax in figure 3.7 on the following page to the proposed clausal syntax in figure 3.8 on the next page. The **@ModelDefinitions** annotation allows more than one **@Model** annotation to attach to the class since Java syntax does not allow for more than one type

of annotation to attach to any Java construct. While this is cumbersome, adhering to pure Java 5 syntax allows syntax checkers in IDEs to signal an error if the array or string is malformed, something not available with the current JML syntax.

```
public abstract class IntList {

  //@ public model non_null int [] elements;

  /*@ public model int average() {
    @   int total = 0;
    @   for (int i = 0; i < elements.length; i++) {
    @     total += elements[i];
    @   }
    @   return total / elements.length;
    @ }
    @ */

}
```

Figure 3.7 Model Method Base Example

```
@ModelDefinitions({
  @Model("public @NonNull int [] elements;"),
  @Model("public int average() {"
       + "  int total = 0;"
       + "  for (int i = 0; i < elements.length; i++) {"
       + "    total += elements[i];"
       + "  }"
       + "  return total / elements.length;"
       + "}")
})
public abstract class IntList {}
```

Figure 3.8 Model Method Clausal Example

3.4.3 Specification Cases

Finally, a major component of heavyweight JML specifications is the specification case. Figure 3.9 on the following page contains a simple example in the regular syntax while figure 3.10 on the next page shows the example in the originally proposed clausal syntax.

```
public abstract class IntList {

    //@ public model non_null int [] elements;

    /*@   public normal_behavior
      @     requires 0 <= index && index < elements.length;
      @     ensures \result == elements[index];
      @ also
      @   public exceptional_behavior
      @     requires index < 0 || index >= elements.length;
      @     signals (ArrayIndexOutOfBoundsException e)
      @             index < 0 || index >= elements.length;
      @*/
    public abstract /*@ pure @*/ int elementAt(int index)
                                 throws ArrayIndexOutOfBoundsException;

}
```

Figure 3.9 Specification Case Base Example

```
@Model("public @NonNull int [] elements;")
public abstract class IntList {

    @Also({
      @SpecCase(visibility = Visibility.PUBLIC,
                specCaseType = SpecCaseType.NORMAL_BEHAVIOR,
                requires = "0 <= index && index < elements.length;",
                ensures = "#result == elements[index];"),
      @SpecCase(visibility = Visibility.PUBLIC,
                specCaseType = SpecCaseType.EXCEPTIONAL_BEHAVIOR,
                requires = "index < 0 || index >= elements.length;",
                signals = "(ArrayIndexOutOfBoundsException e)"
                        + "index < 0 || index >= elements.length;")
    })
    public abstract @Pure int elementAt(int index)
                         throws ArrayIndexOutOfBoundsException;

}
```

Figure 3.10 Specification Case Clausal Example

It would be shorter to specify the specification case annotations as constructs like **@NormalBehavior** or **@ExceptionalBehavior** that inherit from a common annotation **@SpecCase**, but this is not possible since Java does not allow annotation inheritance. Instead, I use the header field technique from the **model** and **invariant** cases instead, as shown in figure **??** on page ??. This reduces the number of fields in the **@SpecCase**, making it easier to specify and read.

```
@Model("public @NonNull int [] elements;")
public abstract class IntList {

    @Also({
      @SpecCase(header = "public normal_behavior",
                requires = "0 <= index && index < elements.length;",
                ensures = "#result == elements[index];")
      @SpecCase(header = "public exceptional_behavior",
                requires = "index < 0 || index >= elements.length;",
                signals = "(ArrayIndexOutOfBoundsException e)"
                        + "index < 0 || index >= elements.length;")
    })
    public abstract @Pure int elementAt(int index)
                          throws ArrayIndexOutOfBoundsException;

}
```

Figure 3.11 Specification Case Clausal Example (with strings)

CHAPTER 4. IMPLEMENTATION

Other than language design, the majority of the work took place in the compiler. This section covers adapting the JML compiler to parse Java 5 annotations, adding the additional compiler pass to process the JML annotations, and translating the annotations for insertion into the JML abstract syntax tree[1].

4.1 Prework

Current versions of the JML compiler only handle Java 2 syntax and do not fully support Java 5 constructs like annotations or generics. The first stage of work involved changing the compiler syntax to include Java 5 annotations and changing the bytecode reader and writer to allow for the new format.

4.1.1 Resolving Syntax Tree Issues

The current implementation of JML defines its grammar using the ANTLR parser-generator [23] and is based off of a public-domain Java 2 grammar found on ANTLR's website. In order to start the move up to Java 5, David Cok used an updated version of the same public-domain grammar and integrated JML constructs into the existing syntax. However, since Cok's compiler did not include the JML runtime assertion checker (necessary for processing such assertions as **`requires`**), it could not be used to implement this project, so I used the existing JML2 compiler instead.

For the most part, this integration was smooth. Only one problem persisted which I ignored in order to advance the work. In order to allow users to declare an annotation definition, the Java 5 language requires this definition to be prefaced by **`@interface`**. Since all other annotations can also appear in exactly the same place as this annotation, ANTLR throws a nondeterminism warning between **`@interface`** and `@ identifier`. Other parser implementations took care of this by adding lookahead in this location, but adding a lookahead statement to the JML compiler caused token source code

[1] Some of the code segments in this chapter are shortened from the actual implementation for clarity

locations to be incorrect. Since JML unit tests make sure that error messages point to the correct token in errors, adding the lookahead made tests fail. I left this problem unresolved since it would be easier to design a new compiler from scratch than try to figure out the source of this problem. Revamping the compiler is out of the scope of this thesis.

4.1.2 Writing Bytecode

The other major error that came up was in importing classes compiled in Java 5 into the modified JML2 compiler. The compiler complained that it found an unknown symbol 'e' in the bytecode. According to a revision of the second edition of the Java Virtual Machine Specification, the symbol 'e' stands for an enumeration type [35]. Integrating it into the bytecode reader took care of the problem.

4.2 Adding an Additional Compiler Pass

The next stage of the work involved processing the strings inside of the annotations, which required an extra compiler pass.

4.2.1 Resolving Annotation Types

In practice, not all annotations in a Java class would be JML annotations. To distinguish JML annotations from other annotations with the same simple name, the compiler must resolve an annotation's full package name before processing the annotations. Since the first pass of the JML compiler takes care of this, the new compiler pass must occur after this pass. However, once the compiler adds all of the new additions to its syntax tree, it needs to run the annotations through the first pass as well, requiring it to backup and reprocess. Instead of doing this, the added compiler pass additionally resolves the annotation packages before continuing its processing, as shown in figure 4.1.

```
Set annotations = self.annotations();
for (Iterator iter = annotations.iterator(); iter.hasNext(); ) {
  JAnnotation annotation = (JAnnotation) iter.next();
  annotation.checkType(context);
}
```

Figure 4.1 Partial Package Resolution

This piece of code gets the annotations from the code construct being processed (say a method or class) and calls `annotation.checkType(context)` to tell each annotation to check and store its full package name, according to the context created and stored by the compiler. This allows future methods to filter annotations by their package names and only process JML annotations.

4.2.2 Walking the Abstract Syntax Tree

The JML compiler regularly uses the *visitor pattern* [9] in order to walk the syntax tree, so it is necessary to include only one class to perform this weaving. The class takes the compilation unit as an argument, brakes it up into its separate parts, and calls the appropriate visitors. The visitor pattern makes this process much easier since the visitors are already included in all necessary classes. All that had to be added in some places were some public accessors so that the visitor could access the different constructs. One such example of this is in the class `JMethodDeclaration`, which stores the annotations from the parser and allows indirect access them by way of an unmodifiable `Set`, as shown in figure 4.2. Thus, any other class with an instance of this class can access the annotations but not have the ability to change them.

```
public Set annotations() {
  return Collections.unmodifiableSet(annotations);
}
```

Figure 4.2 Annotation Accessor in JMethodDeclaration.java

4.3 Translating the Annotations

The final stage of work involved reading the annotations and integrating them into the abstract syntax tree.

4.3.1 Writing the Parser

The current compiler parses the entire compilation unit, both the Java syntax and the JML syntax included in comments. While using the new annotations requires parsing all of the Java expression grammar plus the added JML expression grammar, it is necessary to start parsing using arbitrary meth-

ods from the parser, based on the type of annotation being processed. For example, when parsing a **@Requires** annotation, the program calls `jmlRequiresClause[..]` in the parser, shown in figure 4.3. It parses the specific type of predicates that can appear inside of a **requires** clause (such as `jmlPredicate[]` or `#not_specified`) and returns an instance of `JmlRequiresClause` containing the partial abstract syntax tree.

```
jmlRequiresClause [TokenReference sourceRef, boolean redundantly]
returns [JmlRequiresClause self = null]
{ JmlPredicate predicate = null; }
:
( predicate = jmlPredicate[] | "#not_specified" )
SEMI
{ self = new JmlRequiresClause(sourceRef, redundantly, predicate); }
;
```

Figure 4.3 Grammar for `jmlRequiresClause[]`

I originally thought I could extend the original `Jml.g` grammar with the new `Jml5.g` grammar, but this caused nondeterminism errors. Specifically, figure 4.4 shows an abbreviated version of the original JML grammar

```
jmlSpecClause[] : (jmlRequiresClause[] | jmlEnsuresClause[] | ...) ;

jmlRequiresClause[] : "requires" jmlPredicate[] SEMI ;

jmlEnsuresClause[] : "ensures" jmlPredicate[] SEMI ;
```

Figure 4.4 Partial JML Grammar

while figure 4.5 shows what the abbreviated JML5 grammar would look like when overridden.

```
jmlSpecClause[] : (jmlRequiresClause[] | jmlEnsuresClause[] | ...) ;

jmlRequiresClause[] : jmlPredicate[] SEMI ;

jmlEnsuresClause[] : jmlPredicate[] SEMI ;
```

Figure 4.5 Partial JML5 Grammar

The problem is with `jmlSpecClause[]`. In the original JML grammar, the `"requires"` and

`"ensures"` clauses keep the parser from having to guess which production to jump to. In JML5, these are moved to **`@Requires`** and **`@Ensures`** and eliminated from the parsing grammar. However, the parser no longer can differentiate between the two productions because of the same starting production, `jmlPredicate[]`. I solved this problem by taking `jmlSpecClause[]` completely out of the grammar by rewriting `Jml5.g` to not extend `Jml.g`.

An alternate way to approach this problem is to simply call `jmlPredicate[]` for all of the annotations that need it. However, this approach is very complicated when considering parsing a more complicated expression, such as `jmlRepresentsClause[]` in figure 4.6. Parsing through syntax like `(ASSIGN | L_ARROW)` is much more suited for a parser-generator than it is for handwritten code, so it is easier and more consistent to call only one entry parser method per annotation.

```
jmlRepresentsClause [..] :
  jmlStoreRefExpression[]
  ( (ASSIGN | L_ARROW) jmlSpecExpression[] | "#such_that" jmlPredicate[] )
  SEMI
  ;
```

Figure 4.6 Grammar for `jmlRepresentsClause[]`

4.3.2 Integrating the Parsed Elements

At the beginning of the compilation pass, the visitor class walks through the abstract syntax tree, calling various helper methods that use the parser to construct the partial abstract syntax tree for the JML expressions. A single method contains the call to the parser, relying on method names and reflection to call the appropriate parser method, as shown in figure 4.7 on the next page.

This method was the fastest and most general way to construct the parser, but it has its problems. This technique is so general that in the event of an error, it can only return **`null`** since it has no knowledge of the type of class in the abstract syntax tree it should be returning. Also, the compiler does not halt at this point since it is collecting errors to display to the user at the end of the full compilation.

The more type-safe way would be to construct a parser method for each call into the parser, since each method called in the parser returns a specific type to place into the abstract syntax tree. While factory methods could be used to construct the parser, each parser method needs the same boilerplate

```
private Object parse(TokenReference token, String inputString,
  String method, Object[] arguments, Class[] parameterTypes) {
  ...
  try {
    parser = new Jml5Parser(..);
    Method rule
      = parser.getClass().getDeclaredMethod(method, parameterTypes);
    return rule.invoke(parser, arguments);
  } catch(Exception e) {
    compiler.reportTrouble(e);
  }
  return null;
}
```

Figure 4.7 Parser Method

error handling code, so placing these calls into separate methods requires a lot of code duplication. Neither the *template* nor *strategy* design patterns work here because each method returns a different type. The only solution that might work would be to use generics, which would allow each parser method to exist in a separate class which replaces a generic type `T` with the return type of the parser method. However, at the time of this writing, there is no full generics support in the JML2 parser.

The rest of the methods in the visitor class are helper methods, designed to parse each annotation separately, based on its structure. Most of the code here involves parsing through the actual annotation structure, and while there are tools to assist with this process, they are specifically designed for Java-only syntax trees. JML uses wrapper classes around each of the classes representing Java types, making it necessary to build a separate annotation walker.

For example, methods are represented initially in the class `JMethodDeclaration`, shown in figure 4.8 on the following page. Since JML adds a lot more features to the Java language, such as a `JmlMethodSpecification`, it uses the wrapper class `JmlMethodDeclaration` to contain both the initial `JMethodDeclaration` instance and JML-only `JmlMethodSpecification` feature instance. In order to allow visitors to access both of these features, `JmlMethodDeclaration` contains accessors to fields in the delegee class (`JMethodDeclaration`) and direct access to the method specification class (`JmlMethodSpecification`). Figure 4.9 on the next page shows an abbreviated section of `JmlMethodDeclaration`.

```
public class JMethodDeclaration extends JMemberDeclaration {

  public long modifiers() {
    return modifiers;
  }

  public Set annotations() {
    return Collections.unmodifiableSet(annotations);
  }

  public void setModifiers(long modifiers) {
    this.modifiers = modifiers;
  }

}
```

Figure 4.8 JMethodDeclaration.java

```
public class JmlMethodDeclaration extends JmlMemberDeclaration {

  public long modifiers() {
    return delegee.modifiers();
  }

  public Set annotations() {
    return delegee.annotations();
  }

  public void setModifiers(long modifiers) {
    delegee.setModifiers(modifiers);
  }

  public JmlMethodSpecification methodSpecification() {
    return methodSpec;
  }

  public void setMethodSpecification(JmlMethodSpecification methodSpec) {
    this.methodSpec = methodSpec;
  }

}
```

Figure 4.9 JmlMethodDeclaration.java

When the visitor visits an instance of `JmlMethodDeclaration`, as shown in figure 4.10, it uses the `filterAnnotations(..)` method to preprocess the annotations, placing the JML-only annotations into a `Map` using the annotation name as its key. At this point, it calls methods like `translateModifierAnnotation(..)` and `translateSpec(..)` to determine the necessary information to parse inside the annotation and use that information to call the parser and return the resulting partial abstract syntax tree. Then the visitor can use the accessors to add the partial abstract syntax tree to the existing one.

```
public void visitJmlMethodDeclaration(JmlMethodDeclaration self) {
  Map jmlAnnotations = filterAnnotations(self.annotations());

  long modifiers = translateModifierAnnotations(jmlAnnotations);
  if (modifiers != 0) {
    self.setModifiers(self.modifiers() | modifiers);
  }

  JmlSpecification spec = translateSpec(jmlAnnotations);
  if (spec != null) {
    self.setMethodSpecification(spec);
  }
}
```

Figure 4.10 Visiting a `JmlMethodDeclaration`

4.4 Conclusion

This technique requires a lot of repetitive code, since each helper method returns a different type, based on the annotation it processes. Building a separate annotation walker would have assisted in reducing the code in this process, but since this is a prototype, it is easier and faster to just copy and paste new sections of code. However, it does eliminate the complicated comment parsing involved in the normal JML2 parser, since the final `Jml5.g` grammar does not inherit from the original grammar. Also, this technique would be much simpler when basing the JML compiler off of an existing compiler with better abstract syntax tree access. This would eliminate lot of the helper code in JML involved in parsing through annotations in the syntax tree since it would be delegated to the existing compiler.

CHAPTER 5. CASE STUDIES

To explore the initial use of the new JML annotations, I placed them inside of several test cases, designed to both check how they looked and how the parser performed.

5.1 Two Small Test Cases

The marker and single element annotations were the easiest to program and thus were the first to have test cases designed for them. I compared each of these test cases to how they should report errors with the current JML parser.

The first class, `MarkerAnnotations`, tests specifically the **`@Pure`** annotation, as seen in figure 5.1.

```
package org.jmlspecs.checker.testcase.syntax.annotations;

import org.jmlspecs.annotations.*;

public class MarkerAnnotations {

  private @SpecPublic int variable;
  private @SpecPublic int variable2;

  public @Pure int getVariable() {
    return variable;
  }

  public @Pure void setVariable(int variable2) {
    this.variable2 = variable2;
  }

}
```

Figure 5.1 MarkerAnnotations.java

The first method, `getVariable()`, is pure as marked, but the second method, `setVariable(..)`, is not pure as marked since it has the side effect of assigning to `variable2`. The new JML checker

correctly alerts the user to this fact.

The second class, `SingleElementAnnotations`, tests the **`@Requires`** and `Ensures` clauses, since they are the most common clauses to contain parsible predicates, as seen in figure 5.2.

```
package org.jmlspecs.checker.testcase.syntax.annotations;

import org.jmlspecs.annotations.*;

public class SingleElementAnnotations {

  private @SpecPublic int test1 = 0;
  private @SpecPublic int test2 = 0;

  @Requires("test1 >= 0;")
  @Ensures("test1 >= 1;")
  public int increment1() {
    return test1++;
  }

  @Requires("test2 >= 0;")
  @Ensures("test2 >= 1;")
  public int increment2() {
    return test2;
  }

}
```

Figure 5.2 SingleElementAnnotations.java

In the same way, the first method, `increment1()`, has a correct specification, but the second method, `increment2()`, does not increment `test2` as specified. When compiled with the runtime assertion checker and run, the program correctly throws a postcondition error for `increment2()`.

5.2 One Large Test Case

Finally, a large class, `AAList`, tests some of the features used by the advanced specification cases, as shown in figure 5.3 on the next page. This is a fairly simple array-type class, adapted from some of the specifications of other Java collection classes, but it does effectively demonstrate the different normal and exceptional behavior specification cases, as well as displaying some of the non-Java mathematical expressions JML is capable of, such as the two-way implication (**`<==>`**).

The specification here is correct, but a good test to see whether the runtime assertion checker

```
package org.jmlspecs.checker.testcase.syntax.annotations;

import org.jmlspecs.annotations.*;
import org.jmlspecs.models.*;

public class AAList {

  private @SpecPublic Object[] array;
  private @SpecPublic int size;

  @Also({
    @SpecCase(
      header = "public normal_behavior",
      requires = "0 <= initialSize;",
      assignable = "array, size;",
      ensures = "array != null && array.length == initialSize"
                          + "&& this.isEmpty();"),
    @SpecCase(
      header = "public exceptional_behavior",
      requires = "initialSize < 0;",
      assignable = "#nothing;",
      signalsOnly = "IllegalArgumentException;",
      signals = "(IllegalArgumentException) initialSize < 0;")
  })
  public AAList(int initialSize) {
    if (initialSize < 0) {
      throw new IllegalArgumentException("Illegal initial size: "
                                         + initialSize);
    }
    array = new Object[initialSize];
    size = 0;
  }

  @Assignable("array, size;")
  @Ensures("array.length == 10 && size == 0 && this.isEmpty();")
  public AAList() {
    array = new Object[10];
    size = 0;
  }

  @Ensures("#result == size && #result >= 0;")
  public @Pure int size() {
    return size;
  }

  @Ensures("#result <==> (size == 0);")
  public @Pure boolean isEmpty() {
    return size() == 0;
  }

}
```

Figure 5.3 AAList.java

is working is removing the check for a negative `initialSize` from one of the constructors, as in figure 5.4, leaving the specification incorrect.

```
@Also({
  @SpecCase(
    header = "public normal_behavior",
    requires = "0 <= initialSize;",
    assignable = "array, size;",
    ensures = "array != null && array.length == initialSize"
                          + "&& this.isEmpty();"),
  @SpecCase(
    header = "public exceptional_behavior",
    requires = "initialSize < 0;",
    assignable = "#nothing;",
    signalsOnly = "IllegalArgumentException;",
    signals = "(IllegalArgumentException) initialSize < 0;")
})
public AAList(int initialSize) {
  array = new Object[initialSize];
  size = 0;
}
```

Figure 5.4 AAList Incorrect Constructor

Then, initializing the class with a negative `initialSize` and compiling it with the runtime assertion checker throws an exceptional postcondition error, since the check for the negative `initialSize` is absent.

The next chapter will go into more detail about how these annotations can be used in actual applications.

CHAPTER 6. USING JML ANNOTATIONS FOR VERIFICATION OF WEB-BASED DATA ENTRY

Modified from a paper to be submitted to a venue to be determined[1]

Kristina B. Taylor

6.1 Abstract

In the last several years, collaborations in the scientific and medical fields have expanded to include members from around the world, making distributed web-based data entry forms increasingly popular. Data entered in these forms is safety-critical because of its applications in scientific databases, military vehicles, and medical therapy devices. Such safety-critical systems require data type and number range verification on entry to detect errors early and prevent accidents by allowing immediate replacement of error-prone or incorrect data with safe and correct data.

This paper demonstrates the integration of language constructs from the Java Modeling Language (JML), a Java code specification language, with Apache Tapestry-based web forms. Currently, JML tools cannot copy errors found in code specifications to web forms. This new approach allows programmers to write JML specifications for the web form using Java 5 annotations, automatically producing error checking code that highlights entered data not meeting the requirements specification.

6.2 Introduction

Since the Internet has gained popularity as an information medium, companies and organizations have used the web to host services such as online stores, digital libraries, and information search engines. Each one of these services relies on the accurate recording and submission of entered data, but

[1] An early version of this section was written as a class project for Computer Science 515

they are not safety-critical since their failure does not lead to accidents. It is not until these companies use data entry forms for safety-critical applications that data validation requires further research. Several incidents, such as the USS Yorktown failure [30] and the Therac-25 deaths [20], make it clear that data entry and validation is just as important to safety-critical systems as the testing of the system itself.

This paper focuses on one such way to perform input data validation, by marrying web form design with programming language specification languages, such as the Java Modeling Language (JML) [19].

6.3 Significance

To explain the necessity of validating input data, this paper highlights two separate user interface-related accidents. These accidents cover separate areas of input data validation and inspire two scenarios covered later in the paper.

A software glitch caused the first accident, which stranded the USS Yorktown in September 1997. According to Slabodkin, ”the Yorktown lost control of its propulsion system because its computers were unable to divide by the number zero.” The system administrator entered a zero in an input field, causing a database overflow and subsequent system crash. But most disturbing is that Atlantic Fleet officials stated that ”program administrators are trained to bypass a bad data field and change the value if such a problem occurs again” [30].

Bad user interface design caused the second accident when, between June 1985 and January 1987, a radiation therapy machine called the Therac-25 fatally overdosed six people. The device had two modes (x-ray and electron), and the operator was responsible for both selecting one of the modes and entering the radiation dosage. Once it received its instructions from the interface, the machine normally reconfigured its physical configuration based on the entry mode in order to deliver the correct dose and protect the patient from overdose. Because the operator often entered data faster than the designers expected, the machine sometimes operated while the set mode of the device did not correspond to the physical configuration of the system, causing a massive overdose [20].

6.4 Related Work

Form data verification has existed since programmers needed to check the integrity of data intended for use by a database or service, but as demonstrated by the previous examples, this process has been largely ad-hoc. Faulkner and Storey state that while programmers have often used logic and good engineering techniques to double check these data attributes, these techniques are largely omitted from systems. In addition, they note that data-driven systems often need to manipulate and transform entered data, so data verification at the source is essential to the safety of the system. Finally, they assert that the "use of data entry validation is sufficient to demonstrate that adequate data integrity has been attained" [8]. The authors offer many ideas about data entry and transformation verification but do not offer any concrete techniques or data validation software.

Knight, Strunk, Greenwell, and Wasson take data verification one step further, illustrating a technique they used in the Minimum Safe Altitude Warning (MSAW) system, a data-driven safety system used by the U.S. Federal Aviation Administration. They create a data specification to identify places where the data may be malformed or wrong. The specification includes three aspects (data syntax, data context, and data semantics), which identify where errors can propagate in data-driven systems. One advantage the authors notice about data verification was that it has the "potential to mechanically show properties of interest" to inspectors and regulators [17]. Since this specification is designed for extremely complex real-time systems, it does not apply to validating data entered into simple web forms.

Maret, Beney, and Rubel do study entering information into web forms for remote medical systems. They describe a system that takes certain data from the context of the local information system and assists users in filling out forms. Since it depends on local data to assist form completion, it assures that the form data is consistent with already available data [21]. While this helps web form completion, it does not introduce a definitive system for web form validation.

Several web frameworks introduce their own flavor of form validation, but they are all different. Java Server Faces has minimum/maximum value and String length validators, but does not provide parsing for arbitrary Java expressions [33]. Apache Struts requires the user to place form validation in XML files, use a Struts-specific expression language, and generate the error messages manually [1].

While the latter allows message flexibility, using a Struts-specific expression language separated into XML files makes it hard for the user to both learn and use. The Spring Framework goes a step further by allowing Java expressions, but it still requires setup in XML files, which separates the validation from the code [14]. Apache Tapestry, the web framework used in this paper, has form validation capabilities as well, but these do not parse Java expressions [2].

Even though data validation does not have a platform on the web, it does have a platform for regular programming languages in the form of a specification language. The specification language used in this paper, JML, combines several aspects from other specification languages such as Eiffel [22], the Larch language family [12], and VDM [15], and follows in the *design by contract* tradition [4]. At least one group has used JML for web service verification [7], but they focus on verifying the entire system flow and not on data entry validation. Thus, data entry validation using specification languages requires more research to find out whether it is a valid model.

6.5 Approach

This paper bases its proposed approach on the approach taken in JML's runtime assertion checker (RAC). The RAC translates JML into executable methods and inserts them in appropriate places in the code with the help of a bytecode writer. When the user runs the code, these methods check the runtime values expressed in the assertions and alerts the user if any of the assertions fail [5]. While the RAC does this for normal Java code, it does not yet have the capability to write its error-checking code into HTML web forms.

However, some web frameworks, like Apache Tapestry [2] used in this paper, use pure Java classes to implement form logic inside of an HTML form. Tapestry uses two structures to represent a full web page: an HTML template, as in figure 6.1 on the next page and a corresponding Java class, as in figure 6.2 on page 34.

Notice how all of the names of the fields in the template match up with those in the Java class. The only exception is the `form` field, which does not have any direct mapping to the template. Tapestry takes care of this by using the `@Component(id = "login")`, where the `id` field name must match the form id name. This allows insertion of JML code straight into the Java classes in order to record data input errors.

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
  <head><title>Login</title></head>
  <body>
    <t:form t:id="login">
      <t:label for="name"/>
      <input t:type="TextField" t:id="name"/><br/>
      <t:label for="password"/>
      <input t:type="TextField" t:id="password"/><br/>
      <button type="submit">Submit</button>
    </t:form>
  </body>
</html>
```

Figure 6.1 Tapestry Template Example

The JML syntax used in this paper differs from the existing JML syntax, which is normally stored in comments. Some *design by contract* languages, such as Contract4J [37] and Modern Jass [26], use the code annotations introduced in Java 5, which attach textual metadata to code constructs. The author has worked with the creator of Modern Jass to come up with a common set of Java 5 annotations for use in both Modern Jass and JML.

For prototype purposes, the easiest way to insert the assertion code into the Tapestry page class is to use a bytecode writer like Javassist [6], since it allows the programmer to insert bytecode into a class while typing it in normal Java syntax. Javassist understands Java 5 annotations, so it is a good choice for this approach.

6.6 Results

The following sections demonstrate how this approach deals with two separate scenarios. The first scenario involves just checking the bounds on a single field in a form; the second scenario involves checking the bounds based on the interaction of multiple fields in the form.

6.6.1 Checking Individual Properties

This first scenario closely relates to the Yorktown accident, since it deals with data entry in a single field. In addition, the author currently works with scientists to come up with web forms for entering scientific data into databases, so the following example is similar to the code used in that project. The

```
public class Login {

  private Form form;
  private String name, password;

  @Component(id = "login")
  public Form getForm() { return form; }

  public String getName() { return name; }

  public void setName(String name) { this.name = name; }

  public String getPassword() { return password; }

  public void setPassword(String password) { this.password = password; }

}
```

Figure 6.2 Tapestry Java Example

Tapestry Java code for programming such a web form appears as in figure 6.3 (abbreviated for space).

```
public class NewElement {

  @Requires("name.length > 0")
  @Ensures("this.name == name")
  public void setName(String name) { this.name = name; }

  @Requires("number > 0")
  @Ensures("this.number == number")
  public void setNumber(int number) { this.number = number; }

  @Requires("mass > 0")
  @Ensures("this.mass == mass")
  public void setMass(int mass) { this.mass = mass; }

}
```

Figure 6.3 Tapestry Web Form Example

This example introduces the JML language constructs **@Requires** and **@Ensures**. For the method `setMass(..)`, **@Requires**`("mass > 0")` states that, before the method executes, the formal parameter `mass` must be non-zero positive, and **@Ensures**`("this.mass == mass")` states that, after the method executes, the class variable `mass` is equal to the same value as the formal argument `mass`. Web forms are only concerned about the value being set, so looking at the **@Requires** clause is sufficient.

In order to record an error, Tapestry allows the class to declare a variable `form` (see figure 6.2 on the preceding page), which represents the containing form, and record the error on it. This is where the bytecode writer comes into play; it changes the implementation of `setMass(..)` directly in the Java class compiled from figure 6.3 on the previous page to include the code to record the error with the form. Figure 6.4 shows the Java code representation of the modified bytecode.

```
@Requires("mass > 0")
@Ensures("this.mass == mass")
public void setMass(int mass) {
  if (!(mass > 0)) form.recordError("Form requires that mass > 0");
  this.mass = mass;
}
```

Figure 6.4 Transformed `setMass(..)` Method

Since the assertion inside of the **`if`** block comes directly from the **`@Requires`** annotation, the program should automatically insert the entire block into the bytecode itself. At this point the program uses the Javassist library to automatically find loaded classes and, based on several criteria, add additional Java constructs to the method without user interaction. Given a method of type `CtMethod`, Javassist inserts any `String` at the beginning of the method by using the `CtMethod.insertBefore(..)` method. This transforms the methods in the client class shown in figure 6.3 on the previous page into the bytecode version of the example in figure 6.4. After the program makes this change to the class files, Tapestry loads these as page classes and transforms them into a dynamic webpage.

If a user now enters data into the form and forgets to enter the mass data, leaving the value at 0, on submission of the form, the webpage will throw an error, both telling the user where the error is and what values the form will accept, as shown in figure 6.5.

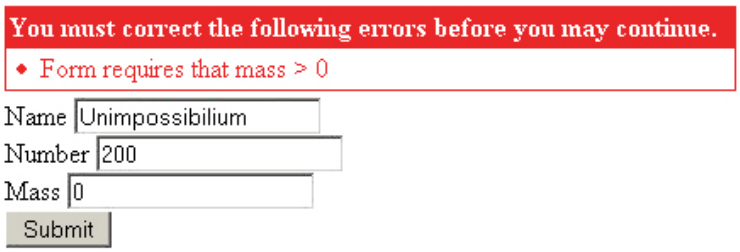

Figure 6.5 Tapestry Error Form

6.6.2 Checking Properties on Form Submit

This second scenario relates to the Therac-25 accidents, since it deals with mode selection on a user interface. Consider the case when multiple fields on a form must agree before the data is valid. In this case, the user must set a certain mode in order for other entered value to be valid. In this case, two of the modes are two chemicals: chemical X and chemical Y. The safe human dosage for chemical X is less than 5 cc; the safe dosage for chemical Y is between 10 and 15 cc. If the user can enter both types of dosage on the same form, then the user must first select the correct chemical and then type in the correct dosage based on the selected chemical.

The approach used in the first scenario does not work here since it assumes that all data fields are independent of each other. However, a valid assumption for this new case is that all data is final when the user submits the form. The program can instead check all values when the user submits the form and throw an error when the check does not succeed. Again, the Tapestry code appears as in figure 6.6.

```
public class NewDose {

  public enum Mode { CHEMICAL_X, CHEMICAL_Y };

  @Requires("mode != null")
  @Ensures("this.mode == mode")
  public void setMode(Mode mode) { this.mode = mode; }

  @Requires("dosage >= 0")
  @Ensures("this.dosage == dosage")
  public void setDosage(int dosage) { this.dosage = dosage; }

  @Requires("(!isModeX() || dosage < 5)" +
         "&& (!isModeY() || (10 < dosage && dosage < 15))")
  public void onSubmitFromNewDose() {
    System.out.println("success");
  }

  private boolean isModeX() { return mode.equals(Mode.CHEMICAL_X); }

  private boolean isModeY() { return mode.equals(Mode.CHEMICAL_Y); }

}
```

Figure 6.6 Multiple Property Web Form

This time, the method `onSubmitFromNewDose()` needs to change in order to check its **`@Requires`**

conditions before it executes the `println(..)` statement. Thus, the bytecode for the new version of `onSubmitFromNewDose()` looks like figure 6.7.

```
@Requires("(!isModeX() || dosage < 5)" +
       "&& (!isModeY() || (10 < dosage && dosage < 15))")
public void onSubmitFromNewDose() {
  if (!((!isModeX() || dosage < 5)
     && (!isModeY() || (10 < dosage && dosage < 15)))) {
    form.recordError("Form requires that " +
         "(!isModeX() || dosage < 5)" +
      "&& (!isModeY() || (10 < dosage && dosage < 15))");
  }
  System.out.println("success");
}
```

Figure 6.7 Transformed `onSubmitFromNewDose(..)` Method

When the user enters data into the form, then the numbers entered depend on the mode that the user has set. If the user types in 12 cc but forgets to switch the mode to chemical Y, then the form alerts the user of the mistake, as in figure 6.8.

Figure 6.8 Tapestry Error Form

6.7 Evaluation

As it is a prototype, this implementation is primitive and needs work. First, the programmer must run a separate program to modify and upload the class files before starting Tapestry. This program must know exactly which file to modify and which form to submit the errors to, both of which are bad assumptions to make. Integrating the program as a service in Tapestry can solve these problems, since Tapestry natively makes several assumptions about where the programmer places files and how the programmer denotes specific fields on the pages. Tapestry loads all page classes from a specified

package and requires that all fields representing forms in the page classes are either named exactly as they are in the corresponding page template or have a `@Component(id = "..")` annotation mapping. Implementing both of these features requires much more research into how Tapestry starts up, for this program must run before Tapestry takes over and starts its own class modification.

Second, the user alert messages are just based on the text in the **@Requires** annotation and do not assist the user much in correcting the error. The normal JML implies syntax is not currently available to this new annotation syntax, so the programmer must manually translate the statement `@Requires("isModeX() ==>dosage < 5")` to its Java-readable logical equivalent. Also, Javassist's documentation states that it does not handle Java 5 enumerations yet, so the programmer must use helper methods like `isModeX()` in place of `mode.equals(Mode.CHEMICAL_X)`, which is much more readable.

One unanswered question is whether running JML's static analysis tools on the form will find other errors during the programming process. For example, there are **@Pure** annotations on the getter methods, which imply that the method only returns the requested value and has no side effects. JML can check for purity and alert the programmer whether the method is really pure or not. This prevents unwanted side effects, such as writing to a database during a call to a getter method instead of waiting until the program has verified all values. Other JML constructs could assist in similar ways, making the form behave exactly like the specifications, but this is a topic for future research.

The most obvious disadvantage is that this technique only currently works with the Tapestry framework. Other web frameworks that are not based on Tapestry have different ways of viewing pages and executable code, so using JML on them may not work at all. Each framework would require a separate tool to mixin JML to its web structure. Thus, one possible area of future work is to design a common tool that allows JML error-checking to integrate into the validation code of any arbitrary web framework.

6.8 Conclusion

These two scenarios demonstrate that this method is promising and that further research is necessary for using JML constructs in web forms. Form validation is necessary to protect the end users of the entered data, and this is only possible when the validation is formalized enough to take advan-

tage of specification checkers like JML. Writing web forms in this way makes sure that programmers write according to the specification and that the program automatically alerts the user to data entry inconsistencies.

CHAPTER 7. DISCUSSION AND SUMMARY

Finally, to conclude, I answer the evaluation questions introduced in the introduction and make my conclusions based on these criteria.

7.1 Discussion

In the introduction, I introduced two questions which my thesis would hopefully answer:

- Is there a way to write JML Java 5 annotations such that they are consistent, readable, usable, and extensible?
- What changes should be made to Java 5 annotations, if any, to make them more useful with JML?

The next two sections answer these in light of the discoveries in this thesis.

7.1.1 Annotations as JML

Table 7.1 on the following page shows various comparisons between the current implementation (JML2) and the proposed Java 5 annotation implementation (JML5). Several criteria (such as method of expression, code placement, and parsing method) have already been covered in previous sections of this thesis, but other portions of the table need more explanation. Familiarity refers to how much of the syntax a newcomer to JML would be expected to know; the familiar annotation syntax for JML5 makes it easier for newcomers to write in JML and not confuse `@`-delimited JML2 comments with Java 5 annotations. Native Java parsing refers to how much of the JML syntax is added to the Java syntax tree when using a pure Java parser. When the JML syntax is defined in Java 5 annotations, more of the JML syntax is in the Java syntax tree. This makes it easier to fulfill the native IDE support criteria, or the amount of content assistance an IDE can supply when writing the JML assertions. While there

Table 7.1 JML2 vs. JML5

Criteria	JML2	JML5
Method of expression	Comments	Java 5 annotations
Code placement	Everywhere	Any declaration (not on statements)
Parsing method	Pure ANTLR parser	Hybrid ANTLR parser and annotation walker
Familiarity	Comment syntax	Fully-typed annotation syntax
Native Java parsing	Comments	Annotation types
Native IDE support	None	Annotation type resolution
Multiple assertions	Yes	Only with annotation array types
Reliance on strings	No	Yes, with delimiters

is no complete native IDE support for JML expressions, since they are included in strings, the support is much better than it is with comments. Multiple assertions refers to whether multiple clauses of the same type can be placed on a single language construct. This is possible in JML2 since it is expressed in comments, but the only way to accomplish this in JML5 is to use separate array types. Finally, JML2 expressions are not expressed in strings while JML5 expressions are, possibly causing problems down the road when having to delimit strings appearing in the assertions themselves.

In addition, the examples throughout the thesis show that while certain annotations in JML5 (`@Pure`) are shorter then their JML2 counterparts (`/*@ pure @*/`), the bigger annotations in JML5 (like `@SpecCase`) are slightly longer than equivalent JML2 listings, simply because of the added complication of the array types for containing multiple annotations of the same type.

Thus, even though chapter 3 showed that there is a consistent, readable, usable, and extensible way to place JML inside of Java 5 annotations, there is no clear cut yes-or-no answer to the question on whether Java 5 annotations should be used to replace the comments in the current JML syntax. There are definite advantages to placing more of the JML syntax into the Java parser; the annotation syntax eliminates the complicated comment parsing technique, stores JML in a more structured form in the Java abstract syntax tree, takes advantage of commonly-known Java syntax, and includes partial native IDE support. However, difficulty writing multiple assertions and relying on strings are big disadvantages to the technique. While there is a temptation to ”get on the annotation bandwagon” as others have, my recommendation is to support both formats until the community decides which one is superior.

7.1.2 Annotations as Java Syntax

The major complaint about the Java 5 annotation syntax is with its lack of flexibility. Since Java does not allow more than one annotation type attached to any language construct, the new syntax must use separate annotations containing an array of a certain type of annotation. There is no inheritance either with annotations, so it is not possible to have a single container annotation for multiple annotations in a specification case. Having either of these features would the design shorter and easier to use. Even though JSR 308 does not attempt to solve this problem, its proposal includes two possible solutions for the multiple annotation problem: desugaring multiple annotations into a normal container annotations and adding new methods to access multiple annotations [34]. Since C# allows multiple attributes to be attached to any language construct [38], the implementation of C# attributes is slightly better than the current implementation of Java 5 annotations. Making Java 5 on par with the implementation of C# would make defining JML assertions via Java 5 annotations much easier.

7.2 Summary

This thesis has shown that since the inception of annotations in Java 5, groups have created small to medium-size design by contract libraries and checkers using these annotations. The goal of this thesis was to determine whether these annotations could effectively express the larger design by contract language of JML and determine what features, if any, needed to be added to Java to make this process simpler. It explained the reasoning behind the final proposed design, evaluating it using four criteria: consistency, readability, usability, and extensibility. It highlighted the procedures used to implement the design into the parser, and it also evaluated the design by testing it both on several small Java classes and on a larger web-form verification test case.

I have concluded that the grammar overviewed in the thesis is the best way to express JML inside of Java 5 annotations, but the actual syntax support in Java 5 has a long way to go before it makes the actual process of writing JML in annotations significantly better than using comments. Even so, if Java finally allows multiple annotations, it would tip the balance in strong favor of using these new annotations. Further work and full evaluation by the community of the prototype constructed for this thesis is required before deciding on one technique over another.

APPENDIX. ANNOTATION SYNTAX

Many descriptions listed here are paraphrased from the JML Reference Manual [19]. Special thanks to Johannes Rieken for the ideas for formatting this section.

`@Accessible`

Description

Specifies what locations a method can read during its execution

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@Also`

Description

Contains a list of specification cases

Arguments

- *value*: **`@SpecCase`**`[]` – **default** {}

`@Assignable`

Description

Specifies what locations a method can assign to before it finishes its execution

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@Axiom`

Description

States that a theorem prover should assume the given predicate is true

Arguments

- *value*: `String` – **default** `None (required)`

`@AxiomDefinitions`

Description

Contains a list of **`@Axiom`** annotations

Arguments

- *value*: **`@Axiom`**`[]` – **default** {}

`@Callable`

Description

Specifies what methods may be called by the current method

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@Captures`

Description

Specifies which references may be stored by the method after it returns

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@Constraint`

Description

Describes how values should change over time

Arguments

- *header*: `String` – **default** **`""`**
- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@ConstraintDefinitions`

Description

Contains a list of **`@Constraint`** annotations

Arguments

- *value*: **`@Constraint`**`[]` – **default** {}

`@Diverges`

Description

Specifies when a method may loop forever

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@Duration`

Description

Specifies the maximum processing time for a method for a specific specification case

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@Ensures`

Description

Specifies that the method that it is attached to guarantees `value` after its execution

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@ForAll`

Description

States that the following specification case must hold for all possible values of the given list of variables

Arguments

- *value*: `String` – **default** `None (required)`

`@Ghost`

Description

Declares a specification-only field whose value is declared in its initialization

Arguments

- *value*: `String` – **default** `None (required)`

`@GhostDefinitions`

Description

Contains a list of **`@Ghost`** annotations

Arguments

- *value*: **`@Ghost`**`[]` – **default** $\{\}$

`@Helper`

Description

Used on a private method or constructor, stating that it is not in any invariants or history constraints

Arguments

None

`@In`

Description

Specifies that the declared field should belong in the given data group(s)

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@InDefinitions`

Description

Contains a list of **`@In`** annotations

Arguments

- *value*: **`@In`**`[]` – **default** {}

`@Initially`

Description

States that each non-helper constructor for each concrete subtype of the enclosing type (including that type itself, if it is concrete) must establish the given predicate

Arguments

- *header*: `String` – **default** **`""`**
- *value*: `String` – **default** `None (required)`

`@InitiallyDefinitions`

Description

Contains a list of **`@Initially`** annotations

Arguments

- *value*: **`@Initially`**`[]` – **default** {}

`@Invariant`

Description

States that the given predicate is true for every visible state

Arguments

- *header*: `String` – **default** `""`
- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@InvariantDefinitions`

Description

Contains a list of **`@Invariant`** annotations

Arguments

- *value*: **`@Invariant`**`[]` – **default** {}

`@Maps`

Description

Describes elements of a data group that are determined dynamically, through a field reference or an array index, or a field of an array index

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@MapsDefinitions`

Description

Contains a list of **`@Maps`** annotations

Arguments

- *value*: **`@Maps`**`[]` – **default** {}

`@Model`

Description

Declares a specification-only field or method whose value can be changed by a **`@Represents`** clause

Arguments

- *value*: `String` – **default** `None (required)`

`@ModelDefinitions`

Description

Contains a list of **`@Model`** annotations

Arguments

- *value*: **`@Model`**`[]` – **default** {}

`@Monitored`

Description

States that a thread must hold the lock on the field before it executes a read or write

Arguments

None

`@MonitorsFor`

Description

States that a thread must hold the lock on all objects in the given list before it executes a read or write on the given field

Arguments

- *header*: `String` – **default** `""`
- *value*: `String` – **default** `None (required)`

`@MonitorsForDefinitions`

Description

Contains a list of **`@MonitorsFor`** annotations

Arguments

- *value*: **`@MonitorsFor`**`[]` – **default** {}

`@NonNull`

Description

Specifies that the field cannot ever be set to **`null`**

Arguments

None

`@NonNullByDefault`

Description

Specifies that, by default, the field cannot ever be set to **`null`**

Arguments

None

`@Nullable`

Description

Explicitly states that the field can be set to **`null`**

Arguments

None

`@NullableByDefault`

Description

Implicitly states that the field can be set to **`null`**

Arguments

None

`@Old`

Description

Allows for the given variables to stand for their initial values in the specification case

Arguments

- *value*: `String` – **default** `None (required)`

`@Pure`

Description

Specifies that the method cannot perform any side effects in its operation

Arguments

None

`@Readable`

Description

States that a given predicate must be true before a given field can be read

Arguments

- *header*: `String` – **default** **`""`**
- *value*: `String` – **default** `None (required)`

`@ReadableDefinitions`

Description

Contains a list of **`@Readable`** annotations

Arguments

- *value*: **`@Readable`**`[]` – **default** {}

`@Represents`

Description

Assigns a value to a given **`@Model`** field, subject to optional restrictions

Arguments

- *header*: `String` – **default** **`""`**
- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@RepresentsDefinitions`

Description

Contains a list of **`@Represents`** annotations

Arguments

- *value*: **`@Represents[]`** – **default** {}

`@Requires`

Description

Specifies that the method needs `value` to be true before the entry into the method

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@Signals`

Description

Specifies the exceptional or abnormal postconditions

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@SignalsOnly`

Description

Specifies the exceptional or abnormal postconditions and what exceptions may only be thrown

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@SpecCase`

Description

Defines a single specification case

Arguments

- *header*: `String` – **default** `""`
- *forall*: `String` – **default** `"#not_specified"`
- *old*: `String` – **default** `"#not_specified"`
- *requires*: `String` – **default** `"#not_specified"`
- *requiresRedundantly*: `String` – **default** `"#not_specified"`
- *ensures*: `String` – **default** `"#not_specified"`
- *ensuresRedundantly*: `String` – **default** `"#not_specified"`
- *signals*: `String` – **default** `"#not_specified"`
- *signalsRedundantly*: `String` – **default** `"#not_specified"`
- *signalsOnly*: `String` – **default** `"#not_specified"`
- *signalsOnlyRedundantly*: `String` – **default** `"#not_specified"`
- *diverges*: `String` – **default** `"#not_specified"`
- *divergesRedundantly*: `String` – **default** `"#not_specified"`
- *when*: `String` – **default** `"#not_specified"`
- *whenRedundantly*: `String` – **default** `"#not_specified"`
- *assignable*: `String` – **default** `"#not_specified"`
- *assignableRedundantly*: `String` – **default** `"#not_specified"`
- *accessible*: `String` – **default** `"#not_specified"`
- *accessibleRedundantly*: `String` – **default** `"#not_specified"`
- *callable*: `String` – **default** `"#not_specified"`
- *callableRedundantly*: `String` – **default** `"#not_specified"`
- *measuredBy*: `String` – **default** `"#not_specified"`
- *measuredByRedundantly*: `String` – **default** `"#not_specified"`
- *captured*: `String` – **default** `"#not_specified"`
- *capturedRedundantly*: `String` – **default** `"#not_specified"`
- *workingSpace*: `String` – **default** `"#not_specified"`
- *workingSpaceRedundantly*: `String` – **default** `"#not_specified"`
- *duration*: `String` – **default** `"#not_specified"`
- *durationRedundantly*: `String` – **default** `"#not_specified"`

`@SpecProtected`

Description

Specifies that this feature is **`protected`** for specification purposes

Arguments

None

`@SpecPublic`

Description

Specifies that this feature is **`public`** for specification purposes

Arguments

None

`@Uninitialized`

Description

Specifies that this feature is to be considered uninitialized

Arguments

None

`@WorkingSpace`

Description

Specifies maximum amount of heap space used by a method

Arguments

- *redundantly*: **`boolean`** – **default** `false`
- *value*: `String` – **default** `None (required)`

`@Writable`

Description

States that a given predicate must be true before a given field can be written to

Arguments

- *header*: `String` – **default** **`""`**
- *value*: `String` – **default** `None (required)`

`@WritableDefinitions`

Description

Contains a list of **`@Writable`** annotations

Arguments

- *value*: **`@Writable`**`[]` – **default** {}

BIBLIOGRAPHY

[1] APACHE SOFTWARE FOUNDATION. Struts. From `http://struts.apache.org/` (Date retrieved: March 19, 2008), 2008.

[2] APACHE SOFTWARE FOUNDATION. Tapestry 5. From `http://tapestry.apache.org/tapestry5/` (Date retrieved: March 19, 2008), 2008.

[3] BARTETZKO, D., FISCHER, C., MOLLER, M., AND WEHRHEIM, H. Jass - Java with assertions. In *Workshop on Runtime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01* (2001). Published in *Electronic Notes in Theoretical Computer Science*, K. Havelund and G. Rosu (eds.), 55(2), 2001.

[4] BURDY, L., CHEON, Y., COK, D. R., ERNST, M. D., KINIRY, J. R., LEAVENS, G. T., LEINO, K. R. M., AND POLL, E. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer 7*, 3 (June 2005), 212–232.

[5] CHEON, Y., AND LEAVENS, G. T. A runtime assertion checker for the Java Modeling Language (JML). In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA, June 24-27, 2002* (June 2002), H. R. Arabnia and Y. Mun, Eds., CSREA Press, pp. 322–328.

[6] CHIBA, S. Javassist - a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java* (Oct. 1998).

[7] DESMET, L., PIESSENS, F., JOOSEN, W., AND VERBAETEN, P. Bridging the gap between web application firewalls and web applications. In *FMSE '06: Proceedings of the fourth ACM workshop on Formal methods in security* (New York, NY, USA, 2006), ACM, pp. 67–77.

[8] FAULKNER, A., AND STOREY, N. Data: An often-ignored component of safety-related systems. In *Proceedings of the MOD Equipment Assurance Symposium (ESAS02)* (Bristol, UK, 2002), MOD.

[9] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[10] GHEZZI, C., AND MONGA, M. Fostering component evolution with C# attributes. In *Proceedings of the International Workshop on Principles of Software Evolution* (Orlando, Florida, 2002), ACM, pp. 22–28.

[11] GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[12] GUTTAG, J. V., HORNING, J. J., AND WING, J. M. The Larch family of specification languages. *IEEE Software 2*, 5 (Sept. 1985), 24–36.

[13] HAGGARD, W. nContract – creating configurable run-time contract verification for .NET components. Master's thesis, Virginia Polytechnic Institute and State University, Mar. 2005.

[14] INTERFACE21. Spring Framework. From `http://www.springframework.org/` (Date retrieved: March 19, 2008), 2008.

[15] JONES, C. B. *Systematic Software Development Using VDM*, second ed. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., 1990.

[16] KATRIB, M., LEDESMA, E., AND PANEQUE, L. Including assertions in .NET assemblies. *.NET Developer's Journal 1*, 9 (Sept. 2003).

[17] KNIGHT, J. C., STRUNK, E. A., GREENWELL, W. S., AND WASSON, K. S. Specification and analysis of data for safety-critical systems. In *22nd International System Safety Conference* (2004), N. Welch and A. Boyer, Eds., International Systems Safety Society.

[18] KOLBLY, D. M. *Extensible Language Implementation*. PhD thesis, University of Texas, Austin, Dec. 2002.

[19] LEAVENS, G. T., POLL, E., CLIFTON, C., CHEON, Y., RUBY, C., COK, D. R., MÜLLER, P., KINIRY, J., AND CHALIN, P. JML Reference Manual. Available from `http://www.jmlspecs.org`, Oct. 2007.

[20] LEVESON, N. *Safeware : System Safety and Computers*. Addison-Wesley Pub Co., Reading, Mass., 1995.

[21] MARET, P., RUBEL, P., AND BENEY, J. Multimedia information interchange: Web forms meet data servers. In *1999 IEEE International Conference on Multimedia Computing and Systems* (Los Alamitos, CA, USA, 1999), vol. 02, IEEE Computer Society, p. 499.

[22] MEYER, B. *Object-oriented Software Construction*, second ed. Prentice Hall, New York, NY, 1997.

[23] PARR, T. J., AND QUONG, R. W. ANTLR: A predicated-LL(k) parser generator. *Software—Practice & Experience 25*, 7 (1995), 789–810.

[24] RED HAT MIDDLEWARE, LLC. Hibernate. From `http://www.hibernate.org/` (Date retrieved: March 19, 2008), 2008.

[25] RESOLVE CORPORATION. eXtensible C#. From `http://www.resolvecorp.com/products.aspx` (Date retrieved: March 19, 2008), 2008.

[26] RIEKEN, J. Contract by design for java - revised. Master's thesis, Universität Oldenburg, 2007.

[27] ROYER, M., ALAGIĆ, S., AND DILLON, D. Reflective constraint management for languages on virtual platforms. *Journal of Object Technology 6*, 10 (Nov.-Dec. 2007), 59–79.

[28] SIMONYI, C. The death of programming languages, the birth of intentional programming. Tech. rep., Microsoft, Inc., Sept. 1995. Available from `http://citeseer.nj.nec.com/simonyi95death.html`.

[29] SJÖGREN, A., CRNKOVIC, I., AND LARSSON, M. A method for support for design by contract on the .NET platform. Tech. rep., Department of Software Engineering, Mälardalen University, 2002.

[30] SLABODKIN, G. Software glitches leave Navy Smart Ship dead in the water. *Government Computer News* (1998).

[31] SUN MICROSYSTEMS, INC. JSR 175: A metadata facility for the Java programming language. From `http://jcp.org/en/jsr/detail?id=175` (Date retrieved: March 19, 2008), 2004.

[32] SUN MICROSYSTEMS, INC. JSR 305: Annotations for software defect detection. From `http://jcp.org/en/jsr/detail?id=305` (Date retrieved: March 19, 2008), 2006.

[33] SUN MICROSYSTEMS, INC. JavaServer Faces Technology. From `http://java.sun.com/javaee/javaserverfaces/` (Date retrieved: March 27, 2008), 2007.

[34] SUN MICROSYSTEMS, INC. JSR 308: Annotations on java types. From `http://jcp.org/en/jsr/detail?id=308` (Date retrieved: March 19, 2008), 2007.

[35] SUN MICROSYSTEMS, INC. The class file format. From `http://java.sun.com/docs/books/jvms/second_edition/ClassFileFormat-Java5.pdf` (Date retrieved: March 19, 2008), 2008.

[36] THOMSCHKE, S. OVal – the object validation framework for Java 5 or later. From `http://oval.sourceforge.net/` (Date retrieved: March 19, 2008), 2007.

[37] WAMPLER, D. Contract4J for design by contract in Java: Design pattern-like protocols and aspect interfaces. In *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (2006), Y. Coady, D. H. Lorenz, O. Spinczyk, and E. Wohlstadter, Eds., Published as University of Virginia Computer Science Technical Report CS–2006–01, pp. 27–30.

[38] WILTAMUTH, S., AND HEJLSBERG, A. C# language specification. From `http://msdn.microsoft.com/library/en-us/csspec/html/CSharpSpecStart.asp` (Date retrieved: April 2, 2003), Dec. 2002.

[39] WYK, E. V. Domain specific meta languages. In *Proceedings of the 2000 ACM Symposium on Applied Computing* (Como, Italy, 2000), vol. 2, ACM, pp. 799–803.

ACKNOWLEDGMENTS

I would like to thank all of those who helped me with various parts of this thesis. Gary T. Leavens got me initially interested in JML and language design, helped me organize my research and thesis, introduced me to LaTeX, and taught me that if you have to do something repetitive more than twice, you should let the computer do it. Krishna Rajan gave me interesting applications for this research and how it could be used in seemingly unrelated fields. Hridesh Rajan asked very thoughtful questions and helped me see more potential directions for this research. I would also like to thank all of the JML developers, both at ISU and on the mailing list, for helping me understand JML and evaluating the initial idea. Special thanks goes to Johannes Rieken for collaboration on a common Java 5 annotation language between JML and Modern Jass.

The research described in this thesis was funded in part by NSF grant DMR-0603644.