Discussion of Design Alternatives for JML Java 5 Annotations

Kristina P. Boysen and Gary T. Leavens

January 9, 2008

Abstract

The Java Modeling Language (JML) is a specification language for Java that allows users to specify intended code behavior through assertions attached to the code. Currently, these assertions are written in Java comments in the code. This paper describes a proposed syntax for translating the current JML assertions into new Java 5 annotations. This would allow syntax checkers in tools like Eclipse to check part of the assertion syntax and give code completion assistance to the user.

1 Introduction

This paper results from efforts to consider how to design JML in Java 5 annotations. There are three different ways of designing these annotations to contain the JML predicates. This paper discuss the advantages and disadvantages of each of these.

2 Background

The issue that spawned this discussion was how to deal with modifiers within modifier-based assertions like **`model`**. These assertions can contain other modifiers like **`non_null`** that can also appear separately next to other Java modifiers. Consider the example in figure 1 inspired by one in the JML Reference Manual:

```
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 1: Base Example

This example could be translated into Java 5 annotations in several ways, from an all-encompassing annotation to many smaller annotations. Several criteria were considered when judging how usable these annotations were to the users and designers:

Consistency Annotations may 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 Three Approaches

3.1 The Single Annotation

One extreme technique would be to just put all of JML annotation inside one grand annotation. Projects such as XVP [1] do this by using an encompassing **@Constraint** annotation to hold the entirety of a JML expression. In our examples, we will use **@JML** to minimize confusion with JMLs **constraint** keyword. An example is given in figure 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 2: Single Annotation Example

3.2 The Parameter Annotation

The other extreme technique would be 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 as we will see, going fully by this philosophy causes problems.

```
@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: Parameter Annotation Example

3.3 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 could occur by itself would have an annotation with that name and retain that structure throughout all expressions, even within strings.

```
@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 4: Clausal Annotation Example

4 Discussion

Now we can evaluate each approach based on the criteria stated above.

4.1 Consistency

The single annotation approach is the most consistent of the three, since all grammar is placed inside of the annotation. 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 because of the addition of these sugars, but since it does not split up annotations into component parts, it is much more consistent than the parameter approach.

4.2 Readability

The clausal approach is the most readable of the three, since it strikes a compromise between the two extreme approaches. For the single annotation approach, having to read **@JML**`(...)` all the time decreases readability since the user must to read past the annotation to figure out the assertion type. For the parameter approach, separating the predicate out too far would also decrease readability since every component of the predicate is separated into different pieces in the annotation. The **signals** clause used 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**. If we use this same strategy for a statement like

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

it translates into

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

We cannot expect to make fields for all of the other modifiers that can attach to **model**. Even though these modifiers are known (see JML Reference Manual, Appendix B), any language redesigns would cause us to need to change this.

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

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

while **model** would turn into

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

a much shorter and more compact annotation.

4.3 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 they are required or not. 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.

4.4 Extensibility

The single annotation and clausal approaches are the most extensible, since most or all of the language extensions would be done in the parser. Thus, the parser would continue to be responsible for the error checking, while the hand-written code would only check the type of the annotation and feed all relevant information back to the parser. In the parameter approach, a language change would require 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.

4.5 Summary

Using these criteria, we conclude that the single annotation approach is not helpful to new users and that the parameter annotation approach is too complicated. Thus, the clausal annotation approach seems the best compromise to us. 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 we see this as the best approach, the rest of the paper focuses on translating various JML language constructs into clausal annotations.

5 Use Cases

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

5.1 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 5 in the regular JML syntax:

```
public abstract class IntList {
    //@ public model non_null int [] elements;

    //@ public invariant elements.length < 10;

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

Figure 5: Model and Invariant Base Example

In the original proposed JML clausal syntax, this would be expressed as in figure 6 on the next page.

```
@Model(modifiers = "public @NonNull", value = "int [] elements;")
@Invariant(modifiers = "public", value = "elements.length < 10;")
public abstract class IntList {
    @Ensures("#result == elements.length;")
    public abstract @Pure int size();
}
```

Figure 6: Model and Invariant Clausal Example

This is a very difficult syntax to work with, specifically because of having to explicitly specify the modifiers as a field. For the **model** field, it would be easiest to specify this as like `@Model("public @NonNull int [] elements")`, but this does not work with **invariant**, since it would look like `@Invariant("public elements.length < 10")`, which doesn't really make sense.

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 above **model** field as

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

while writing the **invariant** specification as

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

since type specifications can only be modified with **public**, **protected**, **private**, 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 **static**). From a semantic point of view, the visibility modifiers are not of the same type as the **static** modifier, making it unreasonable to create a single enumerated type and too complicated to make two types.

5.2 Model Methods

Another advantage of the previous approach is that it makes it easier to specify **model** methods. Consider the example in figure 7 using the **model** method in the original 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 7: Model Method Base Example

In the proposed JML syntax, this would appear as in figure 8 on the next page

```
@ModelDefinitions({
  @Model("public @NonNull int [] elements;"),
  @Model("public model 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 8: Model Method Clausal Example

The **@ModelDefinitions** annotation allows more than one **@Model** annotation to attach to the class, since the Java 5 annotation design does not allow for more than one type of annotation to attach to any Java construct, such as a class or method. While using **@ModelDefinitions** and writing strings is cumbersome, they are necessary for Java 5 annotations. Adhering to pure Java 5 constructs allows syntax checkers in development environments like Eclipse to signal an error if the array or string is malformed, something that is not available when writing the @ symbols in the current JML syntax. This example shows that the clausal approach for **@Model** is flexible enough to encompass methods.

5.3 Specification Cases

A major component of heavyweight JML specifications is the use of specification cases. A simple example is as in figure 9.

```
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_only ArrayIndexOutOfBoundsException;
      @*/
    public abstract /*@ pure @*/ int elementAt(int index);
}
```

Figure 9: Specification Case Base Example

The proposed JML annotation syntax would have appeared as in figure 10 on the next page.

```
@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_only = "ArrayIndexOutOfBoundsException;")
    })
    public abstract @Pure int elementAt(int index);
}
```

Figure 10: Specification Case Clausal Example

Notice the array of **@SpecCase** annotations inside of the **@Also** annotation. It would be shorter to specify the **@SpecCase** annotations as constructs like **@NormalBehavior** or **@ExceptionalBehavior** that inherit from a common annotation **@SpecCase**, but this is not possible since Java 5 annotations do not allow user-defined inheritance. Thus, the array of annotations inside **@Also** must be of the single type **@SpecCase**.

Like in the **model** and **invariant** cases, it is not desirable to have to define an enumeration for every modifier, so a good compromise would be to express all modifiers in the **@SpecCase** as strings, or as in figure 11, which 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_only = "ArrayIndexOutOfBoundsException;")
    })
    public abstract @Pure int elementAt(int index);
}
```

Figure 11: Specification Case Clausal Example (with strings)

6 Conclusion

The clausal approach seems like the most promising one, at least at the current stage. It gives a lot of the error checking responsibility to the parser, which is where it should belong in this case, while making the annotations easier to use for new users and shorter to read. More investigation is needed into how all JML assertions would be expressed with the clausal approach. This technique, at least, covers more assertions than the former technique.

References

[1] Mark Royer, Suad Alagic, and Dan Dillon. Reflective constraint management for languages on virtual platforms. *Journal of Object Technology*, 6(10):59–79, 2007.