Desugaring JML Method Specifications

Arun D. Raghavan and Gary T. Leavens

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

Desugaring JML Method Specifications

Arun D. Raghavan and Gary T. Leavens*
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
arunragh@microsoft.com, leavens@cs.iastate.edu

May 20, 2003

Abstract

JML, which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) designed to specify Java modules. JML features a great deal of syntactic sugar that is designed to make specifications more expressive. This paper presents a desugaring process that boils down all of the syntactic sugars in JML into a much simpler form. This desugaring will help one manipulate JML specifications in tools, understand the meaning of these sugars, and it also allows the use of JML specifications in program verification.

1 Introduction

JML [14], which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) [24] designed to specify Java [1, 9] modules. JML features a great deal of syntactic sugar that is designed to make specifications more expressive [13].

Syntactic sugars are additions to a language that make it easier for humans to use. Syntactic sugar gives the user an easier to use notation that can be easily *desugared*, i.e., translated, to produce a simpler "core" syntax. Desugaring is useful both for manipulating JML specifications in tools and for understanding their meaning. One tool that uses part of this desugaring process is jmldoc [5, 21]. Jmldoc is a tool that makes browsable HTML pages from JML specifications. This tool ships with the JML release.[1]

In this paper we focus mainly on *method specifications*, i.e., on specifications that describe methods. Indeed, we only treat a subset of JML's method specification syntax, because we leave for future work a description of how the desugarings presented here interact with JML's redundancy features and with its refinement calculus features (model programs). JML, and its tools, also inherit and allow refinement of other declarations in classes and interfaces, but those are combined in a straightforward way [8, 13, 14]. Essentially a union is used to combine inherited or refined declarations, and a union, which has the semantics of a conjunction, is used for clauses with predicates, such as invariants and history constraints. Hence inheritance and refinement of these other declarations is not discussed further in this document.

After some more background about JML method specifications, we describe their desugaring in detail.

2 JML Method Specifications

This section describes method specifications, first in general terms, and then by giving their syntax.

2.1 Basic Method Specifications

In JML, a basic method specification, like those in other BISLs [24], is based on the use of pre- and postconditions [10]. A precondition in JML is introduced by the keyword `requires`, and a postcondition is introduced by the keyword `ensures`. The predicates used in pre- and postconditions are written in an extension to a subset of Java's expression syntax, which is restricted to be side-effect free [14]. One example of an extension is that JML allows logical implication to be written as the infix operator "==>". Another example is that the pre-state value of an expression, E, may be written as "`\old(E)`" within a postcondition; this notation is adapted from Eiffel [18].

JML method specifications, however, include much more than just pre- and postconditions. The example given in Figure 1 illustrates the most primitive form of a JML method specification. As shown in this figure, the JML's syntax uses annotation markers (`/*@` and `@*/` [14, Appendix B.2.4]) to delimit text that JML sees. Annotations look like Java comments, and so are ignored by Java. JML reads the text between the annotation markers; however, JML ignores at-signs (`@`) at the beginning of a line within the body of an annotation. The annotation for a method precedes the method itself.

In this example, the keyword `public`, which precedes `behavior` in the specification case, means that this is a specification case intended for clients. It implies that the specification case may not use names that have less than public visibility. The next keyword, "`behavior`" indicates that this is a "heavyweight" specification case, meaning that it is intended to be complete in the sense that omitted clauses have predefined defaults as opposed to being unknown (as in the "lightweight" form of a specification case, where a behavior keyword is omitted) [14]. However, in this example, no clauses are omitted.

Following the keyword behavior are several clauses, each of which starts with a JML-specific keyword (`forall`, `old`, `requires`, etc.). We explain each of these in turn. The JML reference manual [16], gives a more detailed explanation of each clause's semantics.

The `forall` clause is used to declare universally quantified logical variables that can be used throughout the specification case. In the example of Figure 1, the specification must hold for all values of the `int` variable i. (Note that i is only used in the `ensures` clause in this example.)

The second clause in Figure 1 is an `old` variable declaration. This is used to abbreviate side-effect free expressions that can be used throughout the rest of the specification. The values of such expressions are evaluated in the pre-state of the method's execution; that is, the variables can be thought of as initialized just after parameter passing to the values of the expressions. This timing is the explanation for the keyword "old". Like the variables declared in the `forall` clause, the variables declared in the `old` clause are logical variables, and have a scope that extends for the rest of this specification case. In the example, `old_size` is initialized to the old value of the instance field `size` (declared at the bottom of the class in the figure). Note that the `old` clause is distinct from JML's `\old()` expression; the semantic connection is that `\old()` expressions can be given a semantics using `old` declarations.

As mentioned above, the `requires` clause gives the method's precondition. If the precondition is not true in the pre-state, none of the rest of the specification case needs to be satisfied by the implementation. In a heavyweight specification case, an omitted requires clause defaults to one with precondition `true`, and hence no requirements are placed on the caller.

The `diverges` describes pre-states for which a method may go into an infinite loop, or otherwise not return to its caller. Thus the predicate in a `diverges` clause is interpreted in the pre-state (since, if it is satisfied, there need not be a post-state). In a heavyweight specification case, such as that shown in Figure 1, the default for this clause is `false`, hence this clause is normally omitted from JML method specifications. If the predicate in the diverges clause is false, as in the example, and if the specification case's precondition is true, then the method call must finish its execution within a finite time (unless the Java virtual machine encounters a runtime error such as stack overflow or running out of memory [14]). Thus if the specification has `false` as its diverges clause predicate, then it is a total correctness specification case.

The `assignable` clause gives a frame axiom [4] for a specification case. The frame axiom lists the locations that the method may assign to, and under what conditions it may assign to them. Given a specific pre-state, only the locations mentioned whose conditions are true in that pre-state, or that are not depended on by such locations [20], may be assigned to by the method's execution, all other

```
//@ import org.jmlspecs.models.*;

public class NonNegStack {

    /*@ public behavior
      @   forall int i;
      @   old int old_size = size;
      @   requires 0 <= size && size < MAX_SIZE;
      @   diverges false;
      @   assignable size if elem >= 0, elems[*] if elem >= 0;
      @   accessible size if elem >= 0;
      @   when true;
      @   working_space 0 if elem >= 0;
      @   duration 20 if elem >= 0;
      @   ensures \old(elem >= 0)
      @         && \result == size && size == old_size + 1
      @         && ((0 <= i && i < old_size) ==> elems[i] == \old(elems[i]))
      @         && elems[size-1] == elem;
      @   signals (Exception e) \old(elem < 0)
      @         && (e instanceof IllegalArgumentException);
      @*/
    public int push(int elem) {
        if (elem >= 0) {
            elems[size++] = elem;
            return size;
        } else {
            throw new IllegalArgumentException("negative");
        }
    }

    public static final int MAX_SIZE = 10;
    private /*@ spec_public @*/ int size = 0;
    private /*@ non_null spec_public @*/ int[] elems = new int[MAX_SIZE];
}
```

Figure 1: An example of the core features of JML method specification cases is given in the specification of the `push` method, which appears just above the code for `push`.

locations may not be assigned to in such an execution of the method. (Note that this does not say that the method must assign to these locations it is permitted to assign to, only that it may do so.) If omitted in a heavyweight specification case, there is no restriction on what locations the method may assign to, so it is usually important to give the frame axiom for a method.

The `accessible` clause says what object fields the method may read during its execution. If omitted in a heavyweight specification case, no restrictions are placed on the implementation, and so normally this clause is omitted from method specifications. The accessible clause lists the object fields the method may read from, and under what conditions it may read from them. Given a specific pre-state, only the locations mentioned whose conditions are true in that pre-state, or that are not depended on by such locations [20], may be read from during the method's execution, all other object fields may not be read from in such an execution of the method. (Again, this just gives permission for the method to read from the named locations, it is not a requirement on the implementation.)

The `when` clause can be used in the specification of concurrency [17]. It says that, assuming that the precondition holds, the method execution only proceeds to completion when the given condition holds; if the condition does not hold, the method's execution waits for a concurrent thread to make it true, and then proceeds.

The `working_space` clause specifies the space that the method call may need to have available on the heap to successfully complete its work. It gives an expression, in bytes, for the maximum additional space needed by the method's execution, when the condition holds in the pre-state [11]. Note that the expression may depend on post-state values. In the example, the method is not allowed to use extra heap space when the argument is non-negative. If omitted, there is no restriction placed on the maximum space that a method may use from the heap during its execution.

The `duration` clause specifies the time that the method call may need to have available on the heap to successfully complete its work. It gives an expression, in Java Virtual Machine (JVM) machine cycles, for the maximum time by the method's execution, when the condition holds in the pre-state [11]. As with the working space, the expression giving the duration may depend on post-state values. In the example, the method is only allowed to use 20 JVM cycles when the argument is non-negative. If omitted, there is no restriction placed on the time that a method may use during its execution.

The `ensures` clause gives a normal postcondition for a method. The JML semantics is that, when the method returns normally, the relationship between the pre-state and the post-state of the method must satisfy the normal postcondition given in the `ensures` clause. In the postcondition for a method with a non-void return type, the notation `\result` denotes the result returned by the method. In a heavyweight behavior specification case, the normal postcondition defaults to `true` if the `ensures` clause is omitted. In the example in Figure 1, the postcondition says that when the method returns normally, it must be the case that `elem` was not negative, and each element of the `elems` that was previously defined has the same value, and the new element added is `elem`, and the result is the post-state value of the stack's size. Since the argument `elem` must be positive for the normal postcondition to be true, the method may not return normally when `elem` is negative.

The `signals` clause is used to say what must be true when a method's execution throws an exception; it gives an exceptional postcondition for the method. In the exceptional postcondition, the exception result (the object thrown in the exception) is denoted by the name in the parenthesized declaration following the keyword `signals`. In the example in Figure 1, the exception result is named `e`. In a heavyweight behavior specification case, the signals clause defaults to "`signals (Exception e) true;`" if the `signals` clause is omitted; this permits the method to throw any exception permitted by the method's signature. In the example in Figure 1, the exceptional postcondition says that when the method throws an exception, it must be the case that `elem` was negative, and that the exception was of type `IllegalArgumentException`. This means that the method may not throw a different kind of exception when the argument `elem` is negative, and that the method may not throw any exception at all when the argument is non-negative.

It is important to understand that a method call may only have one four possible outcomes in the JML semantics. It may:

- terminate normally, in which case the relation between the pre-state and the post-state must satisfy the normal postcondition given by the `ensures` clause,

- throw an exception of type ET, in which case the relation between the pre-state and the post-state must satisfy the exceptional postcondition in all `signals` clauses that name a supertype of ET (including ET itself),

- not return to the caller (e.g., by looping forever or by exiting the program), in which case the `diverges` clause must have held in the pre-state, or

- encounter a Java virtual machine error (which is signaled by throwing an exception that is a subtype of `java.lang.Error`).

In addition to these basic features of flat method specifications, JML has several forms of syntactic sugar that make specifications more expressive [13, 14]. The aim of this paper is to explain these sugars by boiling them down to the features already described in this subsection.

2.2   Extending Specifications

The most interesting and subtle kind of sugar in JML has to do with support for specification inheritance and behavioral subtyping [8, 15, 13]. In brief, a method specification can be extended or refined by another method specification. In JML, each subtype inherits the public and protected specifications of its supertype's public and protected members [8]; this inheritance includes method specifications. Behavioral subtyping requires that the method specifications in a subtype extend the public and protected method specifications from the overridden supertype method's supertypes [9, Section 8.4.6].

JML also allows for the specification of a single type to be split across several files, using refinement. For example, the specifications in `SinglyLinkedList.refines-jml` can refine the specifications in `SinglyLinkedList.java`. All of the method specifications in a refinement extend the corresponding method specifications in the file being refined.

To make it clear to the reader whether a method specification is an extension (as opposed to a specification of a new method) JML uses the keyword "`also`". Such an *extending-specification* must be used for the specification of an overriding or refining method, regardless of whether any of the methods being overridden or refined has a specification. (Furthermore, an extending specification cannot be used when the method does not override or refine an existing method.)

2.3   JML Grammar

Figures 2–5 give the parts of the JML grammar that are necessary for understanding the desugarings presented below. These sections of the grammar quote a restriction of the grammar from appendix B.1.6 of the "Preliminary Design of JML" [14]. The JML grammar is defined using an extended BNF. The following conventions are used in the grammar:

- Nonterminal symbols are written in *italics*.

- Terminal symbols appear in `teletype` font

- Symbols within square brackets ([ ]) are optional.

- An ellipses (. . .) indicates that the preceding nonterminal or group of optional text can be repeated zero or more times.

Figure 2 gives the top-level syntax of method specifications in JML. Figure 3 gives the syntax of lightweight specification cases. The *lightweight* specification case syntax is designed to be easy to use; however, lightweight specification cases are not assumed to be complete. The *generic-spec-case* syntax is also used as part of the most general form of the heavyweight specification case syntax, given in Figure 4. A *heavyweight* specification case begins with one of the behavior keywords (`behavior`, `normal_behavior`, or `exceptional_behavior`); heavyweight specification cases, are assumed to be complete. Finally, the syntax for *spec-var-decls* is given in Figure 5.

method-specification ::= *non-extending-specification* | *extending-specification*
non-extending-specification ::= *spec-case-seq*
spec-case-seq ::= *spec-case* [`also` *spec-case*] . . .
spec-case ::= *generic-spec-case* | *behavior-spec*
extending-specification ::= `also` *spec-case-seq*

Figure 2: Top-level Method specification syntax of JML (ignoring subclassing contracts, redundant specifications, and model programs).

generic-spec-case ::= [*spec-var-decls*] *spec-header* [*generic-spec-body*]
     | [*spec-var-decls*] *generic-spec-body*
spec-header ::= *requires-clause* [*requires-clause*] . . .
generic-spec-body ::= *simple-spec-body* | '`{|`' *generic-spec-case-seq* '`|}`'
generic-spec-case-seq ::= *generic-spec-case* [`also` *generic-spec-case*] . . .
simple-spec-body ::= *simple-spec-body-clause* [*simple-spec-body-clause*] . . .
simple-spec-body-clause ::= *diverges-clause*
     | *assignable-clause* | *accessible-clause* |
     | *when-clause* | *working-space-clause* |
     | *duration-clause* | *ensures-clause* | *signals-clause*

Figure 3: Generic specification cases, used for "lightweight" specification cases.

behavior-spec ::= [*privacy*] `behavior` *generic-spec-case*
     | [*privacy*] `exceptional_behavior` *exceptional-spec-case*
     | [*privacy*] `normal_behavior` *normal-spec-case*
exceptional-spec-case ::= [*spec-var-decls*] *spec-header* [*exceptional-spec-body*]
     | [*spec-var-decls*] *exceptional-spec-body*
privacy ::= `public` | `protected` | `private`
exceptional-spec-body ::= *exceptional-spec-clause* [*exceptional-spec-clause*] . . .
     | '`{|`' *exceptional-spec-case-seq* '`|}`'
exceptional-spec-clause ::= *diverges-clause*
     | *assignable-clause* | *accessible-clause* |
     | *when-clause* | *working-space-clause* |
     | *duration-clause* | *signals-clause*
exceptional-spec-case-seq ::= *exceptional-spec-case* [`also` *exceptional-spec-case*] . . .
normal-spec-case ::= [*spec-var-decls*] *spec-header* [*normal-spec-body*]
     | [*spec-var-decls*] *normal-spec-body*
normal-spec-body ::= *normal-spec-clause* [*normal-spec-clause*] . . .
     | '`{|`' *normal-spec-case-seq* '`|}`'
normal-spec-clause ::= *diverges-clause*
     | *assignable-clause* | *accessible-clause* |
     | *when-clause* | *working-space-clause* |
     | *duration-clause* | *ensures-clause*
normal-spec-case-seq ::= *normal-spec-case* [`also` *normal-spec-case*] . . .

Figure 4: Behavior specification syntax, used for "heavyweight" specification cases.

Figure 5: Specification variable declarations syntax.

3   Desugaring

This section describes the overall desugaring process, at the most abstract level for a method specification.

In the following, let V stand for a visibility level, one of `public`, `private`, `protected`, or empty.

1. Desugar the use of nested *generic-spec-case-seq*, *normal-spec-case-seq*, and *exceptional-spec-case-seq* in specifications from the inside out (see Section 3.1). This eliminates nesting.

2. Desugar lightweight specifications to V `behavior`, where V is the visibility level of the method or constructor being specified. Desugar *spec-case*s beginning with V `normal_behavior` and V `exceptional_behavior` to V `behavior` (see Section 3.2). This leaves only *spec-case*s that begin with V `behavior`.

3. Combine each *extending-specification* (in a subclass or refinement) with the inherited or refined *method-specification*s into a single *method-specification* (see Section 3.3).

4. Desugar each *signals clause* so that it refers to an exception of type `Exception` with a standard name (see Section 3.4).

5. Desugar the use of multiple clauses of the same kind, such as multiple *requires-clause*s (see Section 3.5).

6. Desugar the `also` combinations within the *spec-case*s. (see Section 3.6).

As a way of explaining the semantics of method specifications, one should imagine the above steps as being run one after another. However, tools may wish to use this semantics differently. For example, the jmldoc tool [5, 21] tries to leave specifications in their sugared form by avoiding these steps when it can. On the other hand, the original version of JML's runtime assertion checker [3], used these steps, while the current version [7] instead factors the parts of a specification into various methods, and uses method calls to execute inherited specifications, which avoids the problem of renaming to avoid name clashes inherent in the desugarings [6].

3.1   Desugaring Nested Specifications

There are several forms of specification that allow nesting. This allows one to factor out common declarations and `requires` clauses. Figure 6 gives a simple example. (The annotation markers and at-signs in the concrete syntax are not reflected in the grammar.)

In general, all the clauses in the top *spec-header* are copied into each of the nested *spec-case-seq*s. For example, in Figure 6, the outer requires clause, "`requires x > 0;`", is copied into the nested specification cases. The only problem is that one has to be careful to avoid variable capture. To do this, first rename the specification variables declared in the nested *spec-case-seq* so that they are not the same as any of the free variables in the outer *spec-header*.

In more detail, consider the schematic *normal-spec-case* at the top of Figure 7, which, assuming no renaming is needed to avoid capture, is desugared to the one at the bottom.

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

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

$\Rightarrow$

```
/*@   requires x > 0;
  @    requires x % 2 == 1;
  @    ensures \result == 3*x + 1;
  @ also
  @    requires x > 0;
  @    requires x % 2 == 0;
  @    ensures \result == x / 2;
  @*/
int hailstone(int x);
```

Figure 6: A motivational example for nested specifications, on the left, with a desugaring on the right.

```
forall quantified-var-decl_{0,1}; ... forall quantified-var-decl_{0,f0};
old old-var-decl_{0,1}; ... old old-var-decl_{0,v0};
requires P_{0,1}; ... requires P_{0,rn0};
{|
        forall quantified-var-decl_{1,1}; ... forall quantified-var-decl_{1,f1};
        old old-var-decl_{1,1}; ... old old-var-decl_{1,v1};
        requires P_{1,1}; ... requires P_{1,rn1};
        normal-spec-case-body_1
    also ... also
        forall quantified-var-decl_{k,1}; ... forall quantified-var-decl_{k,fk};
        old old-var-decl_{k,1}; ... old old-var-decl_{k,vk};
        requires P_{k,1}; ... requires P_{k,rnk};
        normal-spec-case-body_k
|}

    ⇒

    forall quantified-var-decl_{0,1}; ... forall quantified-var-decl_{0,f0};
    forall quantified-var-decl_{1,1}; ... forall quantified-var-decl_{1,f1};
    old old-var-decl_{0,1}; ... old old-var-decl_{0,v0}
    old old-var-decl_{1,1}; ... old old-var-decl_{1,v1};
    requires P_{0,1}; ... requires P_{0,rn0};
    requires P_{1,1}; ... requires P_{1,rn1};
    normal-spec-case-body_1
also ... also
    forall quantified-var-decl_{0,1}; ... forall quantified-var-decl_{0,f0};
    forall quantified-var-decl_{k,1}; ... forall quantified-var-decl_{k,fk};
    old old-var-decl_{0,1}; ... old old-var-decl_{0,v0};
    old old-var-decl_{k,1}; ... old old-var-decl_{k,vk};
    requires P_{0,1}; ... requires P_{0,rn0};
    requires P_{k,1}; ... requires P_{k,rnk};
    normal-spec-case-body_k
```

Figure 7: Desugaring nested specifications in a *normal-spec-case*.

```
                                        /*@ public behavior
                                          @   requires places > 0;
                                          @   diverges \not_specified;
                                          @   assignable \not_specified;
                                          @   accessible \not_specified;
    //@ requires places > 0;             @   when \not_specified;
    public int shift(int places);   ⇒    @   working_space \not_specified;
                                          @   duration \not_specified;
                                          @   ensures \not_specified;
                                          @   signals (Exception) \not_specified;
                                          @*/
                                        public int shift(int places);
```

Figure 8: An example of desugaring a lightweight specification, on the left, into a behavior specification, on the right, by filling in defaults.

3.2 Desugaring Normal and Exceptional Specifications

At this point, there are no nested *spec-case*s. We obtain behavior specifications in two steps. First, lightweight specifications are desugared by filling the defaults for omitted clauses [14, Appendix A] (most of which are simply "\not_specified"), and then prefixing the result with V behavior, where V is the same as the visibility of the method being specified. This desugars a lightweight specification to a behavior specification. An example of this process is given in Figure 8.

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

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

which says that no exceptions can be thrown. Similarly, we desugar *spec-case*s beginning with V exceptional_behavior to V behavior by inserting the clause "ensures false;", which says that the method cannot return normally.

This process leaves us with a single non-nested *spec-case* sequence, consisting solely of *spec-case*s that are *behavior-spec*s that use the keyword "behavior". (It is not necessary for the desugaring to fill in the defaults for omitted clauses in such heavyweight specification cases [14, Appendix A], although this could be done as well.) However, these *spec-case*s may have different visibility, and for later processing it is convenient to have the specification cases grouped by visibility level. So, we combine *spec-case*s with the same visibility into a single *behavior-spec*, each of these having a *generic-spec-body* that is a *generic-spec-case-seq*, which contains each of the bodies of the *behavior-spec*s with that privacy level, separated by the "also" keyword. Although there is nesting of a sort here, in each such *behavior-spec*, the *spec-var-decls* and *spec-header* are both empty, and the *generic-spec-body* is either a single *simple-spec-body* or consists of a flat *generic-spec-case-seq*; thus the the nesting is not troublesome. This process leaves us with a *spec-case-seq* of up to four *behavior-spec*s with the form described above, one for each privacy level.

A simple example of this desugaring is shown in Figure 9.

3.3 Desugaring Inheritance and Refinement

Extending specifications augment an *inherited specification* from the specification of the same Java method in the superclass (if any), superinterfaces, and the refined specification (if any). As described in the previous step, the inherited specification has up to four *spec-case*s, one for each of the four visibility levels. This step forms an *augmented specification* from the specification given, called the *augmenting specification*, and the inherited and refined specifications. In the augmented specification there are also up to four *spec-case*s, one for each visibility level, again joined together with "also".

9

```
  /*@   public normal_behavior                      /*@ public behavior
    @      requires !empty();                          @ {|
    @      ensures \result == theElems.header();        @   requires !empty();
    @ also                                              @   ensures \result == theElems.header();
    @   public exceptional_behavior                     @   signals (Exception e) false;
    @      requires empty();              ⇒            @  also
    @      signals (EmptyException e) true;             @   requires empty();
    @*/                                                 @   ensures false;
  public Object top() throws EmptyException;            @   signals (EmptyException e) true;
                                                        @ |}
                                                        @*/
                                                      public Object top() throws EmptyException;
```

Figure 9: Example desugaring of `normal_behavior` and `exceptional_behavior`, on the left, to `behavior`, on the right, and then grouping the resulting behavior specifications.

Inherited specification:

> V `behavior`
>   `{|`
> $spec\text{-}case_{A_1}$
>   `also ... also`
> $spec\text{-}case_{A_k}$
>   `|}`

Combined specification:

> V `behavior`
>   `{|`
> $spec\text{-}case_{A_1}$
>   `also ... also`
> $spec\text{-}case_{A_k}$
>   `also`
> $spec\text{-}case_{B_1}$
>   `also ... also`
> $spec\text{-}case_{B_n}$
>   `|}`

Extending specification: ⇒

>   `also`
> V `behavior`
>   `{|`
> $spec\text{-}case_{B_1}$
>   `also ... also`
> $spec\text{-}case_{B_n}$
>   `|}`

Figure 10: Desugaring for extending specifications with `also`.

For each visibility level, V, the corresponding augmented *spec-case* is formed from *generic-spec-cases* from the body of the V `behavior` *spec-case* in the augmenting specification and the *generic-spec-cases* from the body of the corresponding V `behavior` *spec-cases* of superclasses, superinterfaces, and refined specifications. That is, augmenting specifications are combined with the inherited and refined specifications of the same visibility level. The details of how this is done are described below.

The rules for combining specifications from refinements are slightly different from the rules for combining specifications that are inherited from a supertype method specification. In the case of refinements, all four visibility levels of specification are inherited by the extending specification. However, when an inherited specification comes from a supertype, the private visibility specifications are not inherited, and the package-visibility specifications are only inherited when the subtype is in the same package as the corresponding supertype. Thus, when the subtype is in a different Java package than the supertype, only the public and protected specifications are inherited.

An extending specification starts with "`also`" and may have *spec-cases* with each of the four visibilities. For each visibility level, V, we combine the inherited specification with the corresponding *spec-case* of the extending specification, as shown in Figure 10.

```
                                            /*@ public behavior
                                              @    assignable theElements;
                                              @    assignable size;
   /*@ public behavior                        @    ensures \old(size >= 2);
     @    assignable theElements;              @    ensures size() == \old(size - 2);
     @    assignable size;                     @    signals (Exception e)
     @    ensures \old(size >= 2);             @      (e instanceof EmptyException)
     @    ensures size() == \old(size - 2);    @        ==>
     @    signals (EmptyException ee)          @      (\old(empty())
     @        \old(empty())                    @       && ((EmptyException)e) != null);
     @     && ee != null;               ⇒      @    signals (Exception e)
     @    signals (TooSmallException tse)      @      (e instanceof TooSmallException)
     @        \old(size == 1)                  @        ==>
     @     && tse != null;                     @      (\old(size == 1)
     @*/                                       @       && ((TooSmallException)e) != null);
   public void popTwice()                      @*/
     throws EmptyException, TooSmallException; public void popTwice()
                                               throws EmptyException, TooSmallException;
```

Figure 11: Example of the desugaring that standardizes *signals-clause*s.

3.4 Standardizing Signals Clauses

At this point, there may be several *signals-clause*s in each *spec-case*, and each of these may describe the behavior of the method when a different exception is thrown. To standardize these *signals-clause*s, we make each describe the behavior when an exception of type Exception is thrown, and add a check for the originally declared exception's type. We also use a standard, but fresh, name for all exceptions, which will allow the *signals-clause*s to be combined in the next step. (Although the name used must be fresh to avoid capture, we use "e" in the desugaring rules below for conciseness.) This desugaring is done as follows:

signals (*ET n*) *P*; ⇒ signals (Exception e) (e instanceof *ET*) ==> ([((*ET*)e)/n]*P*);

where the notation [((*ET*)e)/n]*P* means *P* with the cast expression ((*ET*)e) substituted for free occurrences of *n*.

An example of this desugaring is shown in Figure 11.

3.5 Desugaring Multiple Clauses of the Same Kind

We next desugar multiple clauses of the same kind by conjoining their predicates within each specification case. Within each specification case this is done for *requires-clause*s, and all of the clauses that can occur in a *simple-spec-body-clause* (see Figure 3).

For the *assignable-clause*s and *accessible-clause*s, this process is purely syntactic, and consists of joining together the lists from each clause with commas. (If "if" and the following predicate are omitted after a *store-ref*, "if true" is used as a default.) For example, the *assignable-clause*s within a specification case are combined shown in Figure 12.

For *working-space-clause*s and *duration-clause*s, multiple clauses of the same kind are combined into a single such clause by first converting the if syntax into a conditional expression, and then taking the minimum of the resulting expressions, the whole expression govrened by an if which is the disjunction of the original conditions.[2] For example, the *duration-clause*s within a specification case are combined as shown in Figure 13 (Again, if the "if" and the following predicate are omitted after a *spec-expression*, "if true" is used as a default.)

For the other clauses, this process conjoins the the predicates in each clause of the same type (within a specification case). For example, for the *requires-clause*s, this is shown schematically in Figure 14. Exactly the same technique is used for the *when-clause*s, *ensures-clause*s, and *diverges-clause*s. For

---

[2] Thanks to David Cok and Steve Edwards for this suggestion.

```
assignable SR_{1,1} if M_{1,1}, ..., SR_{1,n} if M_{1,n};
 ...
assignable SR_{k,1} if M_{k,1}, ..., SR_{k,m} if M_{k,m};
 ⇒
assignable SR_{1,1} if M_{1,1}, ..., SR_{1,n} if M_{1,n}, ..., SR_{k,1} if M_{k,1}, ..., SR_{k,m} if M_{k,m};
```

Figure 12: Combining multiple *assignable-clause*s; the combination of *accessible-clause*s is similar.

```
duration E_1 if DE_1;
duration E_2 if DE_2;
 ...
duration E_{k-1} if DE_{k-1};
duration E_k if DE_k;
 ⇒
duration Integer.min((DE_1 ? E_1 : Integer.MAX_VALUE),
            Integer.min((DE_2 ? E_2 : Integer.MAX_VALUE),
               ...
               Integer.min((DE_{k-1} ? E_{k-1} : Integer.MAX_VALUE),
                          (DE_k ? E_k : Integer.MAX_VALUE)   ) ... ))
         if ((DE_1) || (DE_2) ... || (DE_{k-1})  || (DE_k));
```

Figure 13: The combination of multiple *duration-clause*s; the combination of *working-space-clause*s is similar.

```
requires P_1; ... requires P_n;  ⇒ requires (P_1) && ... && (P_n);
```

Figure 14: Combining multiple *requires-clause*s; the combination of other clauses is similar.

```
signals (Exception e) P_1; ... signals (Exception e) P_n;
 ⇒
signals (Exception e) (P_1) && ... && (P_n);
```

Figure 15: Combining multiple *signals-clause*s.

```
 /*@ public behavior                              /*@ public behavior
   @   assignable theElements;                       @   assignable theElements, size;
   @   assignable size;                              @   duration Integer.min(
   @   duration 2*MC if MC>0;                         @     (MC>0 ? 2*MC, Integer.MAX_VALUE),
   @   duration 50*CT;                                @     (true ? 50*CT, Integer.MAX_VALUE))
   @   ensures \old(size >= 2);                       @    if ((MC>0) || true);
   @   ensures size() == \old(size - 2);             @   ensures \old(size >= 2)
   @   signals (Exception e)                          @    && size() == \old(size - 2);
   @    (e instanceof EmptyException)                 @   signals (Exception e)
   @     ==>                                          @    ((e instanceof EmptyException)
   @     (\old(empty())            ⇒                  @     ==>
   @      && ((EmptyException)e) != null);           @     (\old(empty())
   @   signals (Exception e)                          @      && ((EmptyException)e) != null))
   @    (e instanceof TooSmallException)              @    &&
   @     ==>                                          @    ((e instanceof TooSmallException)
   @     (\old(size == 1)                             @     ==>
   @      && ((TooSmallException)e) != null);        @     (\old(size == 1)
   @*/                                                @      && ((TooSmallException)e) != null));
 public void popTwice()                              @*/
   throws EmptyException, TooSmallException;        public void popTwice()
                                                      throws EmptyException, TooSmallException;
```

Figure 16: Example of the desugaring that eliminates multiple clauses of the same kind within a specification case.

the *signals-clause*s things are marginally more complex, because each signals clause has a bit of extra syntax. However, due to the processing in the previous step, the idea is the same, and is shown in Figure 15.

An example of this desugaring step is given in Figure 16.

After this step, each *generic-spec-case* in the body of each of the up to four *spec-case*s has only one clause of each type, and the *signals-clause*s all describe the behavior of an exception of type Exception with the same name.

3.6   Desugaring Also Combinations

At this point, there is a single method specification, which might have one *spec-case*, for each visibility level. Within each visibility level we can desugar an internal *generic-spec-case-seq* to eliminate the use of "also" (and hence the vestigial nesting) as shown in Figure 17. This process disjoins the preconditions of the various specification cases within a visibility level, and uses implications between each precondition (wrapped in "\old( )" where necessary) and the corresponding predicate, so that each precondition only governs the behavior when it holds [25, 13].

There is one additional detail that needs to be mentioned about Figure 17 — when the specification variable declarations are combined, there is the possibility of variable capture. To avoid capture, one must do renaming in general.

An example of this desugaring is given in Figure 18.

4   Conclusion

We have defined a desugaring of JML's method specification syntax into a semantically simpler form. The end point of this desugaring is suitable for formal study.

One area for future work is to disentangle the various desugaring steps, so that tools could use them in any order. For example, one should be able to desugar multiple clauses of the same kind by using

```
V behavior
{|
    forall-var-decls₁
    old-var-decls₁
    requires P₁;
    diverges T₁;
    assignable WSR₁,₁,₁ if M₁,₁, ..., WSR₁,₁,ₘ₁ if M₁,ₘ₁;
    accessible RSR₁,₁,₁ if N₁,₁, ..., RSR₁,₁,ₘ₁ if N₁,ₘ₁;
    when W₁;
    working_space WSE₁ if W₁;
    duration DE₁ if D₁;
    ensures Q₁;
    signals (Exception e) EX₁;
also ... also
    forall-var-declsₖ
    old-var-declsₖ
    requires Pₖ;
    diverges Tₖ;
    assignable WSRₖ,₁ if Mₖ,₁, ..., WSRₖ,ₘₖ if Mₖ,ₘₖ;
    accessible RSRₖ,₁ if Nₖ,₁, ..., RSRₖ,ₘₖ if Nₖ,ₘₖ;
    when Wₖ;
    working_space WSEₖ if Wₖ;
    duration DEₖ if Dₖ;
    ensures Qₖ;
    signals (Exception e) EXₖ;
|}
```

$\Rightarrow$

```
V behavior
    forall-var-decls₁ ... forall-var-declsₖ
    old-var-decls₁ ... old-var-declsₖ
    requires (P₁) || (P₂) ... || (Pₖ);
    diverges ((P₁) ==> T₁) && ... && ((Pₖ) ==> Tₖ);
    assignable WSR₁,₁ if P₁ && M₁,₁, ..., WSR₁,ₘ₁ if P₁ && M₁,ₘ₁,
               ..., WSRₖ,₁ if Pₖ && Mₖ,₁, ..., WSRₖ,ₘₖ if Pₖ && Mₖ,ₘₖ;
    accessible RSR₁,₁ if P₁ && N₁,₁, ..., RSR₁,ₘ₁ if P₁ && N₁,ₘ₁,
               ..., RSRₖ,₁ if Pₖ && Nₖ,₁, ..., RSRₖ,ₘₖ if Pₖ && Nₖ,ₘₖ;
    when (\old(P₁) ==> W₁) && ... && (\old(Pₖ) ==> Wₖ);
    working_space Integer.min((\old(P₁ && Wₖ) ? WSE₁ : Integer.MAX_VALUE),
                             ..., (\old(Pₖ && Wₖ) ? WSEₖ : Integer.MAX_VALUE) ...)
            if ((P₁ && W₁) || ... || (Pₖ && Wₖ));
    duration Integer.min((\old(P₁ && D₁) ? DE₁ : Integer.MAX_VALUE),
                        ..., (\old(Pₖ && Dₖ) ? DEₖ : Integer.MAX_VALUE) ...)
            if ((P₁ && D₁) || ... || (Pₖ && Dₖ));
    ensures (\old(P₁) ==> Q₁) && ... && (\old(Pₖ) ==> Qₖ);
    signals (Exception e) (\old(P₁) ==> EX₁) && ... && (\old(Pₖ) ==> EXₖ);
```

Figure 17: Desugaring `also` combinations

14

```
                                            /*@ public behavior
  /*@ public behavior                       @   requires (!empty()) || (empty());
  @ {|                                       @   ensures
  @   requires !empty();                     @      (\old(!empty())
  @   ensures \result == theElems.header();  @       ==> \result == theElems.header())
  @   signals (Exception e) false;           @   && (\old(empty())
  @  also                                    @       ==> false);
  @   requires empty();                  ⇒   @   signals (Exception e)
  @   ensures false;                         @      (\old(!empty())
  @   signals (Exception e)                  @       ==> false)
  @      (e instanceof EmptyException)       @   && (\old(empty())
  @      ==> true;                           @       ==> ((e instanceof
  @ |}                                       @               EmptyException)
  @*/                                        @            ==> true));
 public Object top() throws EmptyException;  @*/
                                            public Object top() throws EmptyException;
```

Figure 18: Example of the desugaring that eliminates `also`.

the conjunction rule illustrated in Section 3.5 at any time. Allowing different orderings brings up the possibility that applying the desugaring steps in different orders might produce different results; we should try to prove that this cannot happen.

JML also has several redundancy features [13], for example, `ensures_redundantly` clauses, and the `implies_that` and `for_example` sections in method specifications. These have no impact on most kinds of semantics for JML, and instead serve to highlight conclusions for the reader. They can, however, be used in debugging specifications [22, 23], and the the conditions needed to check them also need to be formally stated (as has been done to some extent for Larch/C++ [12, 13]). More to the point, similar kinds of desugaring apply to at least the `implies_that` and `for_example` sections in method specifications, and these should be investigated more systematically.

More challenging is giving a semantics to JML's model programs, which are derived from the refinement calculus [19, 2].

Acknowledgements

References

[1] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition.* Addison-Wesley, Reading, MA, third edition, 2000.

[2] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Graduate Texts in Computer Science. Springer-Verlag, 1998.

[3] Abhay Bhorkar. A run-time assertion checker for Java using JML. Technical Report 00-08, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, May 2000. Available by anonymous ftp from ftp.cs.iastate.edu or by e-mail from almanac@cs.iastate.edu.

[4] Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[5] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. In Thomas Arts and Wan Fokkink, editors, *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS 03)*, volume 80 of *Electronic Notes in Theoretical Computer Science (ENTCS)*, pages 73–89. Elsevier, June 2003.

[6] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation. Available from `archives.cs.iastate.edu`.

[7] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). Technical Report 02-05, Department of Computer Science, Iowa State University, March 2002. In SERP 2002, pp. 322-328.

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

[9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

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

[11] Joan Krone, William F. Ogden, and Murali Sitaraman. Modular verification of performance constraints. Technical Report RSRG-03-04, Department of Computer Science, Clemson University, Clemson, SC 29634-0974, May 2003.

[12] Gary T. Leavens. Larch/C++ Reference Manual. Version 5.41. Available in `ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz` or on the World Wide Web at the URL `http://www.cs.iastate.edu/~leavens/larchc++.html`, April 1999.

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

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06v, Iowa State University, Department of Computer Science, May 2003. See `www.jmlspecs.org`.

[15] Gary T. Leavens and Krishna Kishore Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, chapter 6, pages 113–135. Cambridge University Press, 2000.

[16] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, and Clyde Ruby. Jml reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, April 2003.

[17] Richard Allen Lerner. Specifying objects of concurrent systems. Ph.D. Thesis CMU-CS-91-131, School of Computer Science, Carnegie Mellon University, May 1991.

[18] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[19] Carroll Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[20] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular specification of frame properties in JML. *Concurrency, Computation Practice and Experience.*, 15:117–154, 2003.

[21] Arun D. Raghavan. Design of a JML documentation generator. Technical Report 00-12, Iowa State University, Department of Computer Science, July 2000.

[22] Yang Meng Tan. Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications. Technical Report 619, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, Mass., June 1994.

[23] Yang Meng Tan. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering*. Kluwer Academic Publishers, Boston, 1995.

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

[25] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.