Desugaring JML Method Specifications

Arun D. Raghavan and Gary T. Leavens

TR #00-03c
March 2000, Revised July, December 2000, August 2001

Keywords: Behavioral interface specification language, formal specification, desugaring, semantics, specification inheritance, refinement, behavioral subtyping, model-based specification, formal methods, precondition, postcondition, Eiffel, Java, JML.

1999 CR Categories: D.2.4 [*Software Engineering*] Software/Program Verification — Formal methods, programming by contract, reliability, tools, JML; D.2.7 [*Software Engineering*] Distribution, Maintenance, and Enhancement — Documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, pre- and post-conditions, specification techniques;

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

July 13, 2000

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 verification.

1 Introduction

JML [10], which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) [17] designed to specify Java [1, 6] modules. JML features a great deal of syntactic sugar that is designed to make specifications more expressive [9].

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 [14]. JMLDoc is an option of the JML checker, which ships with the JML release.[1]

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 JMLDoc, also inherit and allow refinement of other declarations in classes and interfaces, but those are combined in a straightforward way [5, 9, 10]. 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.

*The work of Raghavan and Leavens was supported in part by NSF grant CCR 9803843; the work of Leavens was also supported by NSF grant and CCR-0097907. This paper is a minor adaptation of chapter 2 of Raghavan's Master's thesis [14], which itself was adapted from a previous version of this document.

[1] The JML release is available from `http://www.cs.iastate.edu/~leavens/JML.html`.

2.1 Basic Method Specifications

In JML, a basic method specification, like those in other BISLs [17], is based on the use of pre- and postconditions [7]. 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. 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 writen as "`\old(`E`)`" within a postcondition; this notation is adapted from Eiffel [12].

By default, JML takes a total-correctness view of specifications; when the precondition is true, a 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 [10]). A `measured_by` clause can be used to assist in proofs of termination. In addition, one may also specify, using a `diverges` clause, when a method is allowed to go into an infinite loop.

A method specification can also say when the method may throw various kinds of exceptions. This is done using the `signals` clause. One can visualize the execution of a method call as either:

- terminating normally, in which case the relation between the pre-state and the post-state must be described by the postcondition in the `ensures` clause,
- throwing an exception of type ET, in which case the relation between the pre-state and the post-state must be described by the exceptional postcondition in the `signals` clause that names type ET true,
- looping forever, in which case the `diverges` clause must have held in the pre-state, or
- encountering a Java virtual machine error.

A method specification may also include a frame axiom [4], introduced by the keyword `assignable`, which says what locations it may assign to, and under what conditions it may assign to them. Locations that are not mentioned in the frame axiom may not be assigned to by the method's execution, and the given locations may only be assigned to when the corresponding conditions hold.

Method specifications support two kinds of local declaration. One may declare universally quantified variables in a `forall` clause. Furthermore, one may declare and initialize local specification-only variables for the purpose of highlighting or abbreviating expressions using `old` declarations. The side-effect free initializing expressions in `old` declarations are evaluated in the pre-state (just after parameter passing).

Finally, method specifications have a `when` clause that can be used in the specification of concurrency [11]. A `when` clause says that the method execution only proceeds when the given condition holds.

In addition to these basic features of flat method specifications, JML has several forms of syntactic sugar that make specifications more expressive [9, 10]. 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 [5, 9]. 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 [5]; this inheritance includes method specifications. Behavioral subtyping requires that the method specifications in a subtype extend the public and protected method specifications from the methods it defines which are also in its supertypes.

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 keywords "`also`" and "`and`". One of these forms of

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` *additive-specification* | `and` *conjoinable-spec-seq*
additive-specification ::= *spec-case-seq*

Figure 1: Top-level Method specification syntax of JML.

extending-specification must be used for specification of an overriding or refining method, regardless of whether any of the methods being overridden or refined has a specification. (Furthermore, neither form of extending specification can be used when the method does not override or refine an existing method.) In this paper, extending specifications that begin with the keyword "`also`" are called *additive specifications*, and specifications that begin with "`and`" are called "*conjoinable specifications*."

2.3 JML Grammar

Figures 1–5 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.6 of the "Preliminary Design of JML" [10]. 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 1 gives the top-level syntax of method specifications in JML. Figure 2 gives the syntax of lightweight specifications. The *lightweight* specification syntax is designed to be easy to use; however, lightweight specifications 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 syntax, given in Figure 3. A *heavyweight* specification begins with one of the behavior keywords (`behavior`, `normal_behavior`, or `exceptional_behavior`); heavyweight specifications, also called *behavior specifications*, are assumed to be complete. The conjoinable specification syntax is given in Figure 4; note that there is no *spec-header* in a conjoinable specifcation. Also note that there are both lightweight and heavyweight forms of conjoinable specifications. Finally, the syntax for *spec-var-decls* is given in Figure 5.

3 Semantics

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 `public behavior`. 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`.

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

Figure 2: Generic 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 ::= *assignable-clause* [*assignable-clause*] ...
[*signals-clause*] ... [*diverges-clause*] ...
| *signals-clause* [*signals-clause*] ... [*diverges-clause*] ...
| '`{|`' *exceptional-spec-case-seq* '`|}`'
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 ::= *assignable-clause* [*assignable-clause*] ...
[*ensures-clause*] ... [*diverges-clause*] ...
| *ensures-clause* [*ensures-clause*] ... [*diverges-clause*] ...
| '`{|`' *normal-spec-case-seq* '`|}`'
normal-spec-case-seq ::= *normal-spec-case* [`also` *normal-spec-case*] ...

Figure 3: Behavior specification syntax.

conjoinable-spec-seq ::= *conjoinable-spec* [`and` *conjoinable-spec*] ...
conjoinable-spec ::= *generic-conjoinable-spec* | *behavior-conjoinable-spec*
generic-conjoinable-spec ::= [*spec-var-decls*] *simple-spec-body*
behavior-conjoinable-spec ::= [*privacy*] `behavior` [*spec-var-decls*] *simple-spec-body*
| [*privacy*] `exceptional_behavior` [*spec-var-decls*] *exceptional-simple-spec-body*
| [*privacy*] `normal_behavior` [*spec-var-decls*] *normal-simple-spec-body*
exceptional-simple-spec-body ::= *assignable-clause* [*assignable-clause*] ...
[*signals-clause*] ... [*diverges-clause*] ...
| *signals-clause* [*signals-clause*] ... [*diverges-clause*] ...
normal-simple-spec-body ::= *assignable-clause* [*assignable-clause*] ...
[*ensures-clause*] ... [*diverges-clause*] ...
| *ensures-clause* [*ensures-clause*] ... [*diverges-clause*] ...

Figure 4: Conjoinable specification syntax.

spec-var-decls ::= *forall-var-decls* [*let-var-decls*] | *let-var-decls*
forall-var-decls ::= *forall-var-decl* [*forall-var-decl*] ...
forall-var-decl ::= `forall` *quantified-var-decl* `;`
let-var-decls ::= `old` *local-spec-var-decl* [*local-spec-var-decl*] ...
local-spec-var-decl ::= `model` *type-spec spec-variable-declarators* `;`
| `ghost` *type-spec spec-variable-declarators* `;`

Figure 5: Specification variable declarations syntax.

```
/*@ 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);
```

⇒

```
/*@   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.

3. Combine each *extending-specification* (in a subclass or refinement) with the inherited or refined *method-specification*s, as dictated by the use of `and` or `also` in the syntax, into a single *method-specification* (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 [14] tries to leave specifications in their sugared form by avoiding these steps when it can, but in order to process conjunctive specifications, it does use steps 1–3 when necessary. On the other hand, JML's runtime assertion checker [3] uses all of these steps.

Although the detailed description of the steps that follows describes the process in purely syntactic terms, tools will often manipulate internal representations of the syntax trees instead of their textual form.

3.1 Desugaring Nested Specifications

There are several forms of specification that allow nesting. This is mainly to factor out common declarations and `requires` clauses, but the factoring also applies to the `when` and `measured_by` clauses. Figure 6 gives a simple example. In this figure, the concrete syntax of JML uses annotation markers (`/*@` and `@*/` [10, Appendix B.2.4]), which are used 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. (These at-signs and annotation markers 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*.

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 [10, Appendix A] (most of which are simply "`\not_specified`"), and then prefixing the result with V `behavior`, where V is determined by 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`". 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. 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`".

For each visibility level, V, the corresponding augmented *spec-case* is formed from *generic-spec-case*s from the body of the V `behavior` *spec-case* in the augmenting specification and the *generic-spec-case*s from the body of the corresponding V `behavior` *spec-case*s 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 vary with the form of the augmenting specification, and 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.

```
forall quantified-var-decl_{0,1} ; ... forallquantified-var-decl_{0,f0} ;
old local-spec-var-decl_{0,1} ... local-spec-var-decl_{0,v0}
requires P_{0,1}; ... requires P_{0,rn0};
when W_{0,1}; ... when W_{0,wn0};
measured_by E_{0,1,1} if MB_{0,1,1}, ..., E_{0,1,j_{0,1}} if MB_{0,1,j_{0,1}};  ...
measured_by E_{0,mbn0,1} if MB_{0,mbn0,1}, ..., E_{0,mbn0,j_{0,mbn0}} if MB_{0,mbn0,j_{0,mbn0}};
{|
      forall quantified-var-decl_{1,1} ; ... forallquantified-var-decl_{1,f1} ;
      old local-spec-var-decl_{1,1} ... local-spec-var-decl_{1,v1}
      requires P_{1,1}; ... requires P_{1,rn1};
      when W_{1,1}; ... when W_{1,wn1};
      measured_by E_{1,1,1} if MB_{1,1,1}, ..., E_{1,1,j_{1,1}} if MB_{1,1,j_{1,1}};  ...
      measured_by E_{1,mbn1,1} if MB_{1,mbn1,1}, ..., E_{1,mbn1,j_{1,mbn1}} if MB_{1,mbn1,j_{1,mbn1}};
      normal-spec-case-body_1
   also ... also
      forall quantified-var-decl_{k,1} ; ... forallquantified-var-decl_{k,fk} ;
      old local-spec-var-decl_{k,1} ... local-spec-var-decl_{k,vk}
      requires P_{k,1}; ... requires P_{k,rnk};
      when W_{k,1}; ... when W_{k,wnk};
      measured_by E_{k,1,1} if MB_{k,1,1}, ..., E_{k,1,j_{k,1}} if MB_{k,1,j_{k,1}};  ...
      measured_by E_{k,mbnk,1} if MB_{k,mbnk,1}, ..., E_{k,mbnk,j_{k,mbnk}} if MB_{k,mbnk,j_{k,mbnk}};
      normal-spec-case-body_k
|}

⇒

   forall quantified-var-decl_{0,1} ; ... forallquantified-var-decl_{0,f0} ;
   forall quantified-var-decl_{1,1} ; ... forallquantified-var-decl_{1,f1} ;
   old local-spec-var-decl_{0,1} ... local-spec-var-decl_{0,v0}
       local-spec-var-decl_{1,1} ... local-spec-var-decl_{1,v1}
   requires P_{0,1}; ... requires P_{0,rn0};
   requires P_{1,1}; ... requires P_{1,rn1};
   when W_{0,1}; ... when W_{0,wn0};
   when W_{1,1}; ... when W_{1,wn1};
   measured_by E_{0,1,1} if MB_{0,1,1}, ..., E_{0,1,j_{0,1}} if MB_{0,1,j_{0,1}};  ...
   measured_by E_{0,mbn0,1} if MB_{0,mbn0,1}, ..., E_{0,mbn0,j_{0,mbn0}} if MB_{0,mbn0,j_{0,mbn0}};
   measured_by E_{1,1,1} if MB_{1,1,1}, ..., E_{1,1,j_{1,1}} if MB_{1,1,j_{1,1}};  ...
   measured_by E_{1,mbn1,1} if MB_{1,mbn1,1}, ..., E_{1,mbn1,j_{1,mbn1}} if MB_{1,mbn1,j_{1,mbn1}};
   normal-spec-case-body_1
also ... also
   forall quantified-var-decl_{0,1} ; ... forallquantified-var-decl_{0,f0} ;
   forall quantified-var-decl_{k,1} ; ... forallquantified-var-decl_{k,fk} ;
   old local-spec-var-decl_{0,1} ... local-spec-var-decl_{0,v0}
       local-spec-var-decl_{k,1} ... local-spec-var-decl_{k,vk}
   requires P_{0,1}; ... requires P_{0,rn0};
   requires P_{k,1}; ... requires P_{k,rnk};
   when W_{0,1}; ... when W_{0,wn0};
   when W_{k,1}; ... when W_{k,wnk};
   measured_by E_{0,1,1} if MB_{0,1,1}, ..., E_{0,1,j_{0,1}} if MB_{0,1,j_{0,1}};  ...
   measured_by E_{0,mbn0,1} if MB_{0,mbn0,1}, ..., E_{0,mbn0,j_{0,mbn0}} if MB_{0,mbn0,j_{0,mbn0}};
   measured_by E_{k,1,1} if MB_{k,1,1}, ..., E_{k,1,j_{k,1}} if MB_{k,1,j_{k,1}};  ...
   measured_by E_{k,mbnk,1} if MB_{k,mbnk,1}, ..., E_{k,mbnk,j_{k,mbnk}} if MB_{k,mbnk,j_{k,mbnk}};
   normal-spec-case-body_k
```

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

```
//@ requires places > 0;
public int shift(int places);
```

⇒

```
/*@ public behavior
  @   requires places > 0;
  @   when \not_specified;
  @   assignable \not_specified;
  @   ensures \not_specified;
  @   signals (Exception) \not_specified;
  @   diverges \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.

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

⇒

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

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

Inherited specification:

```
public behavior
{|
  forall-var-decls_1
  old local-spec-var-decl_{1,1} ...
      local-spec-var-decl_{1,v1}
  spec-header_1
  assignable-clause-seq_1
  ensures-clause-seq_1
  signals-clause-seq_1
  diverges-clause-seq_1
 also ... also
  forall-var-decls_k
  old local-spec-var-decl_{k,1} ...
  local-spec-var-decl_{k,vk}
  spec-header_k
  assignable-clause-seq_k
  ensures-clause-seq_k
  signals-clause-seq_k
  diverges-clause-seq_k
|}
```

Extending specification:

```
and
  public behavior
   forall-var-decls_B
   old local-spec-var-decl_{B,1} ...
       local-spec-var-decl_{B,m}
   assignable-clause-seq_B
   ensures-clause-seq_B
   signals-clause-seq_B
   diverges-clause-seq_B
```

⇒

Combined specification:

```
public behavior
{|
  forall-var-decls_1
  forall-var-decls_B
  old local-spec-var-decl_{1,1} ...
      local-spec-var-decl_{1,v1}
      local-spec-var-decl_{B,1} ...
      local-spec-var-decl_{B,m}
  spec-header_1
  assignable-clause-seq_1
  assignable-clause-seq_B
  ensures-clause-seq_1
  ensures-clause-seq_B
  signals-clause-seq_1
  signals-clause-seq_B
  diverges-clause-seq_1
  diverges-clause-seq_B
 also ... also
  forall-var-decls_k
  forall-var-decls_B
  old local-spec-var-decl_{k,1} ...
      local-spec-var-decl_{k,vk}
      local-spec-var-decl_{B,1} ...
      local-spec-var-decl_{B,m}
  spec-header_k
  assignable-clause-seq_k
  assignable-clause-seq_B
  ensures-clause-seq_k
  ensures-clause-seq_B
  signals-clause-seq_k
  signals-clause-seq_B
  diverges-clause-seq_k
  diverges-clause-seq_B
|}
```

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

We desugar *extending-specification*s that use the "`and`" syntax as shown in Figure 10. The idea is that the conjoined specification augments the inherited and refined specifications without changing the precondition (and other parts of the *spec-header*). Hence the desugaring adds each of the clauses in the conjoinable specification to the corresponding sequence of clauses in each *generic-spec-case* of the inherited or refined *generic-spec-body*, as shown in the figure. Furthermore, the *spec-var-decls* from the *extending-specification* are combined as shown, and renaming may need to be done to avoid capture.

In describing this desugaring step, we have assumed that the same kind of desugaring as described above in Sections 3.1 and 3.2 has already been done for the *conjoinable-spec-seq* in the extending specification, resulting in a sequence of up to four *behavior-conjoinable-spec*s, one for each visibility level, each using the keyword "`behavior`", separated by "`and`".

If there is no inherited specification for an *extending-specification* that uses the "`and`" syntax, then the default specification is inherited. The *default specification* is the lightweight specification with default clauses, and is equivalent to a behavior specification with each clause being "`\not_specified`" [10, Appendix A]. The default specification is used in this case because we assume that the person writing the code had some unknown "implicit" specification in mind.

The extending specification using the "`and`" syntax also allows an "`also`" sequence of specifications. After desugaring the "`and`" part, the part following the first "`also`" is desugared as described next.

The other form of extending specification starts with "`also`" and may have *spec-case*s with each

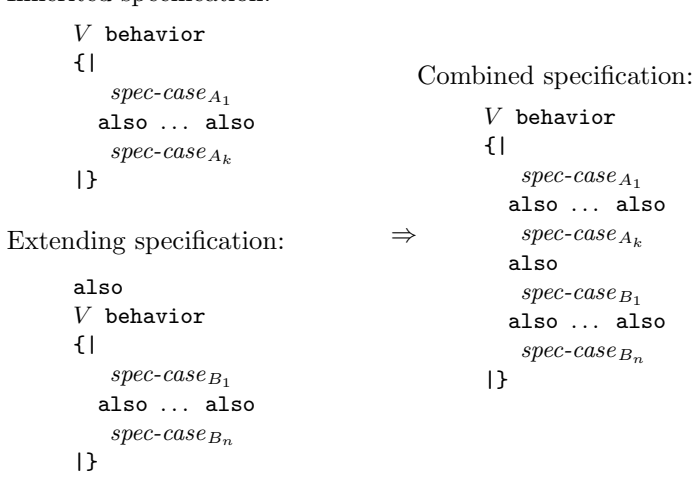

Figure 11: Desugaring for extending specifications with `also`.

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 11.

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 an added check for 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 12.

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, *when-clause*s, *measured-clause*s, *assignable-clause*s, *ensures-clause*s, *signals-clause*s, and *diverges-clause*s.

For the *measured-clause*s and *assignable-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*, the default "`if true`" is used.) For example, the *assignable-clause*s within a specification case are combined as follows.

`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` $M_{k,m}$ `;`

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 as follows:

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

⇒

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

Figure 12: Example of the desugaring that standardizes *signals-clause*s.

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

Exactly the same technique is used for the *when-clause*s, *ensures-clause*s, and *diverges-clause*s. For the *signals-clause*s things are marginally a bit 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 described as follows:

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

An example of this desugaring step is given in Figure 13.

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 14. 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 [18, 9].

There is one additional detail that need to be mentioned about Figure 14. That is that 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 15.

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.

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

⇒

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

Figure 13: Example of the desugaring that eliminates multiple clauses of the same kind within a specification case.

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 type by using 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; it would be nice to prove that this cannot happen.

JML also has several redundancy features [9], for example, `ensures_redundantly`, `implies_that`, and `for_example` clauses. 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 [15, 16], and the the conditions needed to check them also need to be formally stated (as has been done to some extent for Larch/C++ [8, 9]).

More challenging is giving a semantics to JML's model programs, which are derived from the refinement calculus [13, 2]. These seem especially problematic in combination with conjoinable specifications. Thinking about this issue has led us to propose that the conjoinable specifications not be allowed when one of the inherited specifications is a model program.

Acknowledgements

Thanks to Albert Baker, Abhay Bhorkar, Curtis Clifton, Bart Jacobs, K. Rustan M. Leino, Peter Müller, Clyde Ruby, Raymie Stata, and Joachim van den Berg for many discussions about the syntax and semantics of such specifications. Thanks to Curtis Clifton, Don Pigozzi, Clyde Ruby, and Wallapak Tavanapong for comments on earlier drafts.

References

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

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

```
V behavior
{|
    forall-var-decls_1
    old  D_1^*
    requires P_1 ;
    when W_1;
    measured_by E_{1,1} if MB_{1,1}, ..., E_{1,j_1} if MB_{1,j_1};
    assignable SR_{1,1,1} if M_{1,1}, ... SR_{1,1,m_1} if M_{1,m_1};
    ensures Q_1;
    signals (Exception e) EX_1 ;
    diverges T_1;
also ... also
    forall-var-decls_k
    old  D_k^*
    requires P_k ;
    when W_k;
    measured_by E_{k,1} if MB_{k,1}, ..., E_{k,j_k} if MB_{k,j_k};
    assignable SR_{k,1} if M_{k,1}, ... SR_{k,m_k} if M_{k,m_k};
    ensures Q_k;
    signals (Exception e) EX_k ;
    diverges T_k;
|}
```

⇒

```
V behavior
   forall-var-decls_1 ... forall-var-decls_k
   old D_1^* ... D_k^*
   requires (P_1) || (P_2) ... || (P_k) ;
   when (\old(P_1) ==> W_1) && ... && (\old(P_k) ==> W_k) ;
   measured_by E_{1,1} if P_1 && MB_{1,1}, ..., E_{1,j_1} if P_1 && MB_{k,j_1},
           ..., E_{k,1} if P_k && MB_{k,1}, ..., E_{k,j_k} if P_k && MB_{k,j_k} ;
   assignable SR_{1,1} if P_1 && M_{1,1}, ... SR_{1,m_1} if P_1 && M_{1,m_1},
           ..., SR_{k,1} if P_k && M_{k,1}, ..., SR_{k,m_k} if P_k && M_{k,m_k} ;
   ensures (\old(P_1) ==> Q_1) && ... && (\old(P_k) ==> Q_k) ;
   signals (Exception e) (\old(P_1) ==> EX_1) && ... && (\old(P_k) ==> EX_k) ;
   diverges (\old(P_1) ==> T_1) && ... && (\old(P_k) ==> T_k) ;
```

Figure 14: Desugaring `also` combinations

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

⇒

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

Figure 15: Example of the desugaring that eliminates `also`.

[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 Rayomnd Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06p, Iowa State University, Department of Computer Science, August 2001. See `www.cs.iastate.edu/~leavens/JML.html`.

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

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

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

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

[15] 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.

[16] 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.

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

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