Desugaring JML Method Specifications

Arun D. Raghavan and Gary T. Leavens

TR #00-03e
March 2000, Revised July, December 2000, August 2001,
July 2003, May 2005

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 50011-1041, USA

Desugaring JML Method Specifications

Arun D. Raghavan and Gary T. Leavens*
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1041 USA
arunragh@microsoft.com, leavens@cs.iastate.edu

May 27, 2005

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 method specifications more expressive. This paper presents a desugaring process that boils down all of the syntactic sugars in JML method specifications into a much simpler form. This desugaring will help one understand the meaning of these sugars, for example for use in program verification. It may also help manipulation of JML method specifications by tools.

1 Introduction

JML [16], which stands for "Java Modeling Language," is a behavioral interface specification language (BISL) [30] designed to specify Java [1, 11] modules. JML features a great deal of syntactic sugar that is designed to make method specifications more expressive [15].

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 helpful for understanding the meaning of JML method specifications, which may also be helpful in tools [5]. One tool that uses part of this desugaring process is jmldoc [5, 26]. 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 [9, 15, 16]. 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. We start with basic method specifications and then describe extending specifications. The extending

*The work of Raghavan and Leavens was supported in part by NSF grant CCR 9803843; the work of Leavens was also supported by NSF grants CCR-0097907, CCR-0113181, CCF-0428078, and CCF-0429567. This paper is an adaptation of chapter 2 of Raghavan's Master's thesis [26], which itself was adapted from a previous version of this document. More recent versions of this document are largely the work of Leavens, with significant contributions from David R. Cok and others in the JML community.

[1] The JML release is available from `http://www.jmlspecs.org`.

specifications are considered to be syntactic sugars for basic method specifications, and their semantics is ultimately given by desugaring them into basic method specifications.

2.1 Basic Method Specifications

In JML, a basic method specification, like those in other BISLs [30], is founded on the use of pre- and postconditions [12]. 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 [16]. 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 [24].

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 `//@`, see the *JML Reference Manual* [18, Section 4.4]) to delimit text that only the JML tools (but not standard Java tools) see. Annotations look like Java comments, and so are ignored by Java. JML reads the text between the annotation markers. Furthermore, 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 Figure 1, the keyword `public`, which precedes `behavior` in the specification case, means that this is a specification case intended for clients. Using `public` 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) [16]. 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.) and ends with a semicolon. We explain each of these in turn. The *JML Reference Manual* [18], 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 specification expressions (i.e., expressions with JML extensions that are free of side-effects [18]) 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, which is the rationale for using the keyword "`old`". That is, the variables can be thought of as initialized to the values of the corresponding expressions just after parameter passing. 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` clause 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 [16]). Thus if the specification has `false` as its diverges clause predicate, then it is a total

```
//@ 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, elems[*];
      @   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_only IllegalArgumentException;
      @   signals (Exception e) \old(elem < 0)
      @        && \not_assigned(size, elems[*]);
      @*/
    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`.

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. Given a specific pre-state, only locations that are mentioned or are (perhaps indirectly) in data groups that are mentioned [19, 22], may be assigned to by the method's execution; all other 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 the locations that it is permitted to assign to, only that the method may assign to them.) 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 such specification cases.

The `when` clause can be used in the specification of concurrency [23, 27]. It says that, assuming that the precondition holds, the method execution only proceeds to completion at some time when the given condition holds; if the condition does not hold, then the method's execution waits for a concurrent thread to make it true, and then proceeds. (There is no guarantee that the method will proceed the first time this condition holds, so the condition may have to hold many times before the thread may proceed.) If omitted in a heavyweight specification case, its predicate defaults to `true`.

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, following the keyword `if`, holds in the pre-state [13]. Note that the expression may depend on post-state values (including `\result`), so this expression is conceptually evaluated in the post-state, although it may use `\old()` to refer to pre-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 use to successfully complete its work. It gives an expression, in Java Virtual Machine (JVM) machine cycles, for the maximum time available for the method's execution, when the condition, which follows `if`, holds in the pre-state [13]. As with the working space, the expression giving the duration may depend on post-state values (including `\result`), and so is conceptually evaluated in the post-state. 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 this normal postcondition. In the normal 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` array 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 must not return normally when `elem` is negative.

The `signals_only` clause says what types of exceptions may be thrown by a method. The listed exceptions must all be subtypes of `java.lang.Exception`. When the precondition of the specification case holds, the method may only throw instances of types that are not subtypes of `Exception` (typically subtypes of `java.lang.Error`) or exceptions that are subtypes of those listed in the `signals_only` clause. In other words, throwing any exception other than those listed in the `signals_only` clause is prohibited. In a heavyweight specification case, the default list for an omitted `signals_only` clause is the list of exceptions declared in the method's header, or `\nothing` if the method's header does not declare any exceptions. In the example, the `push` method is only allowed to throw an `IllegalArgumentException`.

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 and the `signals_only` clause. 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 both `size` and the elements of `elems` were not assigned. This implies that the method 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 some subtype, ET, of `java.lang.Exception`, in which case the relation between the pre-state and the post-state must satisfy the exceptional postcondition in each `signals` clause that names a supertype of ET (including those that name 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. (JML considers all exceptions that are instances of `java.lang.Throwable` but not instances of `java.lang.Exception` to indicate virtual machine errors. Usually a subtype of `java.lang.Error` is used for this purpose.)

In addition to these features of basic method specifications, JML has several forms of syntactic sugar that make specifications more expressive [15, 16]. 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 [9, 17, 15]. In brief, a method specification can be extended or refined by other method specifications. In JML, each subtype inherits the public and protected specifications of its supertype's visible non-private members [9]; this inheritance includes method specifications. Behavioral subtyping requires that the method specifications in a subtype obey the non-private method specifications from the supertypes that the method overrides [11, Section 8.4.6].

Besides parts of a specification split among different types in a subtyping hierarchy, JML's refinement constructs allow one to split a specification for a single type across several files. 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, without privacy restrictions.

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 subset of the grammar from appendix A.6 of the *JML Reference Manual* [18]. 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.

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

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

lightweight-spec-case ::= *generic-spec-case*
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* | *when-clause*
| *working-space-clause* | *duration-clause* | *ensures-clause*
| *signals-only-clause* | *signals-clause*

Figure 3: Lightweight specification case syntax, which includes *generic-spec-case*.

- 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 *lightweight-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.

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 `non_null` for argument types (see Section 3.1).
2. Desugar the use of `non_null` on result types (see Section 3.2).
3. Desugar the use of `pure` (see Section 3.3).
4. Desugar specifications that are empty to the default specification (see Section 3.4).
5. 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 3.5). This eliminates nesting.
6. 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 3.6). This leaves only *spec-case*s that begin with V `behavior`, for some visibility level V.

heavyweight-spec-case ::= [*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*
| *when-clause* | *working-space-clause* | *duration-clause* | *signals-only-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*
| *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.

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

Figure 5: Specification variable declarations syntax.

7. Combine each *extending-specification* (in a subclass or refinement) with the inherited or refined *method-specification*s into a single effective *method-specification* (see Section 3.7).

8. Desugar each *signals clause* so that it refers to an exception of type `Exception` with a standard name (see Section 3.8).

9. Desugar the use of multiple clauses of the same kind, such as multiple *requires-clause*s (see Section 3.9).

10. For each visibility level, make all the *assignable-clause*s be the same by adding `\only_assigns` to postconditions as needed, and make all the *signals-only-clause*s be the same by adding conjuncts to the exceptional postconditions as needed (see Section 3.10).

11. Desugar the `also` combinations within the *spec-case*s. (see Section 3.11).

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, 26] tries to leave specifications in their sugared form by avoiding these steps when it can. On the other hand, while the original version of JML's runtime assertion checker [3], used an earlier version of these steps, the current version [6, 7] instead factors the parts of a specification into various methods, and uses method calls to execute inherited specifications. Factoring the parts of the specification into various methods avoids the problem of renaming to avoid incorrect variable capture (see below).

3.1 Desugaring Non_Null for Arguments

Use of `non_null` for arguments of a method or constructor adds a precondition that each such argument is not null to the method or constructor's specification. The way this desugaring is accomplished varies, depending on whether the method has any specification.

If the method has no specification, then these preconditions are added as a lightweight specification case, with a preceding `also` if the method overrides some other method. The translation is illustrated for methods in Figure 6. This figure shows a case where only the first and third arguments are specified as `non_null`; in general the translation adds a precondition for each `non_null` argument x, of the form x `!= null`. (The annotation markers and at-signs in the concrete syntax are not shown in such desugarings; in the ISU JML tools they are taken out during lexical analysis.)

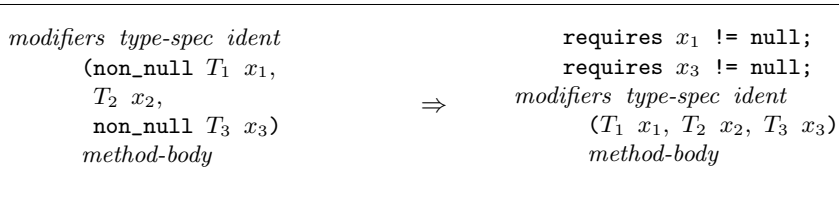

Figure 6: Desugaring for arguments marked as `non_null` in the case where the method has no explicit specification. If this method was an override, then the desugared specification would start with `also`.

However, if the method or constructor has a specification already, then the desugaring is more involved. In this case, the preconditions saying that each `non_null` argument is not null must be added to each individual specification case, as JML's syntax does not allow different kinds of specification cases to be combined into one *spec-case-seq*. So the desugaring uses the auxiliary function, addHeaders, defined in Figure 29 of Appendix A. With this auxiliary function, the desugaring for `non_null` arguments is shown schematically in Figure 7. This figure, shows what happens when the first and third arguments use the modifier `non_null` and the *method-specification* is not an *extending-specification*. If the *method-specification* starts with `also`, this `also` must be preserved. The desugaring for constructors is analogous.

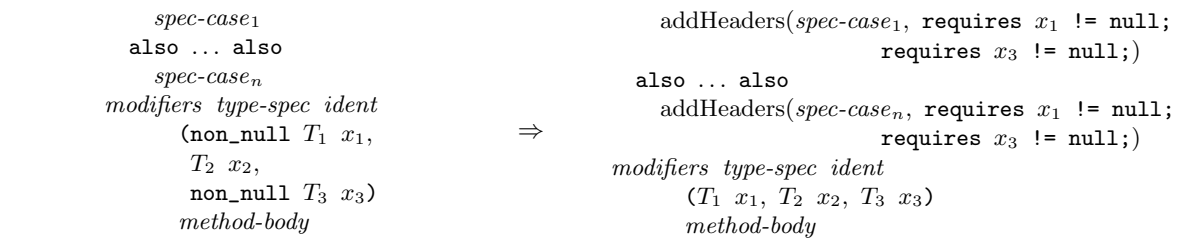

Figure 7: Desugaring for arguments marked as `non_null`, when the method has an explicit specification.

```
/*@ public normal_behavior
  @   requires key.id != null;
  @   ensures theMap.get(key)
  @           == \result;
  @*/
public /*@ pure non_null @*/ Object
 fetch(/*@ non_null @*/ Object key);
```

⇒

```
/*@ public normal_behavior
  @   requires key != null;
  @   {|
  @      requires key.id != null;
  @      ensures theMap.get(key)
  @              == \result;
  @   |}
  @*/
public /*@ pure non_null @*/ Object
  fetch(Object key);
```

Figure 8: Example desugaring `non_null` arguments.

An example of this desugaring is given in Figure 8.
Note that `non_null` on arguments is not inherited, but only affects the specification cases written.

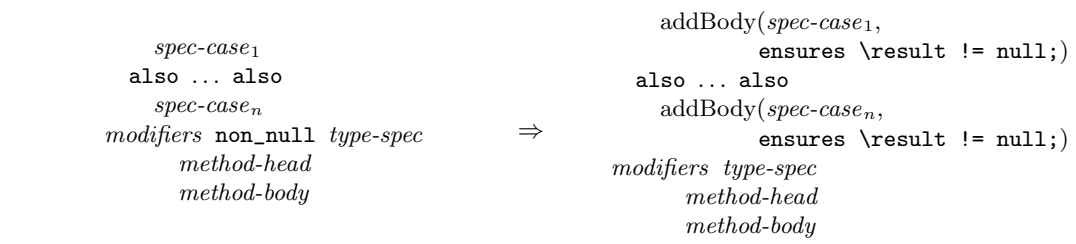

Figure 9: Desugaring for `non_null` results for instance and static methods.

```
/*@ public normal_behavior
  @   requires key != null;
  @   {|
  @      requires key.id != null;
  @      ensures theMap.get(key)
  @              == \result;
  @   |}
  @*/
public /*@ pure non_null @*/ Object
  fetch(Object key);
```

⇒

```
/*@ public normal_behavior
  @   requires key != null;
  @   {|
  @     requires key.id != null;
  @     ensures \result != null;
  @     ensures theMap.get(key)
  @            == \result;
  @   |}
  @*/
public /*@ pure @*/ Object
  fetch(Object key);
```

Figure 10: Example desugaring for `non_null` results.

3.2 Desugaring Non_Null Results

The use of `non_null` for a result type is similar to its use for arguments, in that it adds the normal postcondition `\result != null` to a method specification. The way this desugaring is accomplished is slightly different, depending on whether the method has any specification (thus far).

If the method has no specification, then this postcondition is added as the following lightweight specification case (with a preceding `also` if the method overrides some other method).

```
ensures \result != null;
```

As a consequence of the defaults for lightweight specification cases [16, Appendix A], this lightweight specification case will receive the default precondition of `\not_specified`.

On the other hand, if the method has a specification already (perhaps due to the use of `non_null` for some arguments), then adding this postcondition to the method is more tricky to deal with as a desugaring. The problem is that JML's syntax does not provide an easy way to add a postcondition to a specification case, and adding a new specification case that does not change the method's effective precondition or frame is also difficult to express. Thus we use the auxiliary function, addBody, described in in Figures 30 and 31 of Appendix A, to add the postcondition to every existing specification case for the method.

With this auxiliary function, the desugaring for `non_null` arguments is shown schematically in Figure 9.

An example of this desugaring is shown in Figure 10.

spec-case$_1$
`also ... also`
spec-case$_n$
modifiers$_1$ *V* `pure` *modifiers*$_2$ *type-spec*
method-head
method-body

$\Rightarrow$

addBody(addBody(*spec-case*$_1$, `assignable \nothing;`), `diverges false;`)
`also ... also`
addBody(addBody(*spec-case*$_n$, `assignable \nothing;`), `diverges false;`)
modifiers$_1$ *V* `pure` *modifiers*$_2$ *type-spec*
method-head
method-body

Figure 11: Desugaring for `pure` instance and static methods.

```
/*@ public normal_behavior
  @   requires key != null;
  @   {|
  @      requires key.id != null;
  @      ensures \result != null;
  @      ensures theMap.get(key)
  @              == \result;
  @   |}
  @*/
public /*@ pure @*/ Object
  fetch(Object key);
```

$\Rightarrow$

```
/*@ public normal_behavior
  @   requires key != null;
  @   {|
  @     requires key.id != null;
  @     diverges false;
  @     assignable \nothing;
  @     ensures \result != null;
  @     ensures theMap.get(key)
  @           == \result;
  @   |}
  @*/
public /*@ pure @*/ Object
  fetch(Object key);
```

Figure 12: Example desugaring for `pure`.

3.3 Desugaring Pure

For a constructor, use of `pure` adds the clauses

```
diverges false;
assignable this.*;
```

to each specified *spec-case* for the constructor. These clauses mean that a pure constructor can assign to all instance fields of the class, including all those inherited from superclasses.

If there are no specifications given for the constructor, then the above clauses are added to the method as a lightweight specification. (Hence, if no specifications are given, the implicit precondition is `\not_specified`.)

For an instance or static method (but not for constructors), use of `pure` adds the clauses

```
diverges false;
assignable \nothing;
```

to each given *spec-case* for the method. This is illustrated in Figure 11, which considers the *method-specification* is not an *extending-specification*. If there are no specifications given for the method, then the above clauses are added to the method as a lightweight specification.

An example of this desugaring is shown in Figure 12.

Notice that the translation leaves the modifier `pure` as a modifier on the desugared specification; this is because that modifier is needed to allow the method to be called in assertions, and because it

plays other roles in JML's semantics. In particular, methods that overrride a pure method are also pure, although such overrides do not need to use the `pure` modifier.

Another part of JML's semantics for pure methods and constructors is that JML adds an implicit verification condition that is not part of the method's specification, but which must be satisfied by each implementation of a pure method or constructor, to the effect that it cannot diverge when called. That is, unless a call encounters an error, it must either return normally or throw an exception, even when such a call does not satisfy the method's precondition. For methods, this implicit verification condition is similar to the following specification case [16, Section 2.3.1], except that one must remember that it is only a condition to be verified, and cannot be used be used in verifying even private calls to the method.

```
private behavior
  requires true;
  diverges false;
  assignable \nothing;
```

The verification condition for constructors is analogous, except that, as usual, the assignable clause used for the constructor is "`assignable this.*;`" instead.

3.4 Desugaring Empty Specifications

If at this point a method or constructor has an empty specification, then the specification defaults to one that makes minimal assumptions. For purposes of this desugaring, the specification is considered empty if it has no specification cases, even if it has examples or implications [18]. Note that a specification that uses either `pure` or `non_null` will not be empty.

The desugaring first has to deal with the special case of a completely omitted default constructor for a class. Following this, we discuss empty specifications for methods and constructors that are written in the program.

If the default (zero-argument) constructor of a class is omitted because its code is omitted, then its specification defaults to a lightweight specification consisting of an assignable clause that allows all the locations that the default (zero-argument) constructor of its superclass assigns—in essence a copy of the superclass's default constructor's assignable clause.[2] Such an assignable clause makes the default constructor verify, since it will call the superclass's default constructor. If some other frame is desired, then one has to write the specification, or at least the code, explicitly.

Otherwise, if the method or constructor whose specification is empty does not override another method, then its meaning is taken as that in which all clauses are `\not_specified` [16], which is the same as the following lightweight specification.

```
requires \not_specified;
```

Tools are permitted to make their own interpretations of what `\not_specified` means for various clauses; for example, ESC/Java [10, 21, 8] interprets this to be essentially the same as the lightweight specification "`requires true;`".

However, if the method whose specification is empty overrides some other method, then its meaning is taken to be the following lightweight specification [16].

```
also
  requires false;
```

This somewhat counter-intuitive specification is the unit for `also`-conjunction of specifications [20]. This specification is used so as not to change the meaning of the inherited specification.

[2] This copying from the superclass's constructor's specification is not modular, but JML has no syntax for writing such a frame axiom directly.

```
/*@ 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 13: An example of a nested specification, on the left, with its desugaring, on the right.

3.5 Desugaring Nested Specifications

There are several forms of specification that allow nesting. This allows one to factor out common declarations and `requires` clauses. Figure 13 gives a simple example.

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 13, 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* (i.e., the bound variables) 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 14, 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*.

```
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 14: Desugaring nested specifications in a *normal-spec-case*; note that renaming to avoid capture is needed, but not shown.

3.6 Desugaring Lightweight, Normal, and Exceptional Specifications

At this point, there are no nested *spec-case*s. We eliminate lightweight specification cases and normal and exceptional behavior specification cases in favor of behavior specification cases in this step.

Lightweight specifications are desugared by filling the defaults for omitted clauses [16, 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 15.

Each *spec-case* that starts with V `normal_behavior` is desugared to one that starts with V `behavior` by inserting a `signals` clause of the form "`signals (Exception e) false;`" which says that no exceptions can be thrown.

Similarly, *spec-case*s beginning with V `exceptional_behavior` are desugared 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 *heavyweight-spec-case*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 [16, 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, combine all *spec-case*s with the same visibility into a single *heavyweight-spec-case*. For each visibility, the resulting *heavyweight-spec-case* has a *generic-spec-body* that is a *generic-spec-case-seq*, which contains each of the bodies of the *heavyweight-spec-case*s with that privacy level, separated by the "`also`" keyword. Although there is nesting of a sort here, it is not troublesome. This is because in each such *heavyweight-spec-case* 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*. This process leaves us with a *spec-case-seq* of up to four *heavyweight-spec-case*s with the form described above, one for each privacy level.

A simple example of this desugaring is shown in Figure 16.

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

⇒

```
/*@ public behavior
  @   requires places > 0;
  @   diverges false;
  @   assignable \not_specified;
  @   when \not_specified;
  @   working_space \not_specified;
  @   duration \not_specified;
  @   ensures \not_specified;
  @   signals_only \nothing;
  @   signals (Exception e) \not_specified;
  @*/
public int shift(int places);
```

Figure 15: 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_only EmptyException;
  @*/
public Object top() throws EmptyException;
```

⇒

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

Figure 16: Example desugaring of `normal_behavior` and `exceptional_behavior`, on the left, to `behavior`, on the right, and then grouping the resulting behavior specifications.

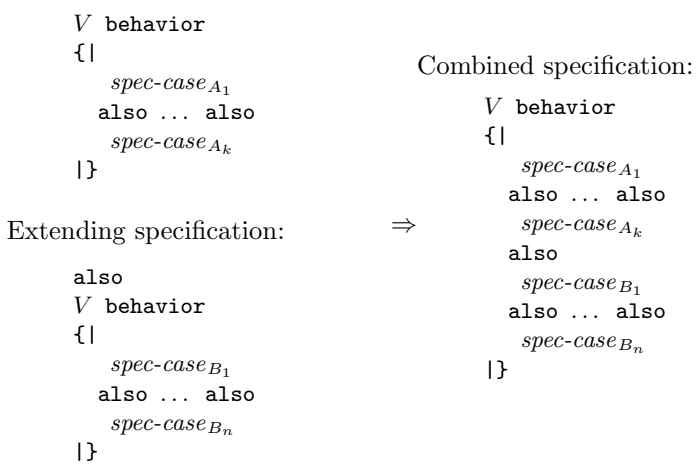

Figure 17: Desugaring for extending specifications with `also`.

3.7 Desugaring Inheritance and Refinement

To desugar inheritance and refinement, we form an *effective specification* that combines the effects of inheritance and refinement; the effective specification is what one would have to write to get the same meaning without using inheritance or refinement.

Each method may refine specifications of the same name and formal argument types from the file that the current file refines. Similarly, each instance method may inherit specifications of instance methods with the same name and argument types in its supertypes (superclass and implemented interfaces). As described in the previous step, for each method, each inherited effective specification has up to four *spec-case*s, one for each of the four visibility levels. In the effective specification that results from combining the extending specification with these inherited specifications there are also up to four *spec-case*s, one for each visibility level, again joined together with "`also`".

An extending specification starts with "`also`" and may have *spec-case*s with each of the four visibilities. For each visibility level, V, we combine the inherited specification with the corresponding *spec-case* of the extending specification, as shown in Figure 17.

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.

In the process of copying inherited assertions from supertypes to subtypes, various kinds of renaming must be done, in general. For example, if a subtype shadows instance fields declared in its supertypes, then some renaming (e.g., changing `f` to `super.f`) is needed, in general, to avoid inadvertent capture of the supertype's field names. Similarly, if the formal parameter names are different in the subtype's overriding method, then renaming must be done on the inherited specifications to substitute that override's formal parameter names. In a refinement, JML requires that all refining specifications for a given method use the same formal parameter names. The JML runtime assertion checker avoids these renaming problems by using method calls to evaluate the inherited parts of a method specification instead of copying predicates with renaming [6, 7].

```
/*@ 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 18: Example of the desugaring that standardizes *signals-clause*s.

3.8 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`;` $\Rightarrow$ `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 18.

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

Figure 19: Combining multiple *assignable-clause*s.

```
signals_only Exception;
signals_only StackPushException;
signals_only IllegalArgumentException, StackException;
 ⇒
signals_only StackPushException;
```

Figure 20: Combining multiple *signals-only-clause*s.

3.9 Desugaring Multiple Clauses of the Same Kind

We next desugar multiple *requires-clause*s and multiple *simple-spec-body-clause*s of the same kind to produce a specification with only one of each such clause. (See Figure 3 for definitions of these nonterminals.)

For the *assignable-clause*s, this process is very simple, and consists of joining together the lists from each clause with commas. The scheme describing how the *assignable-clause*s within a specification case are combined is shown in Figure 19.

For the *signals-only-clause*s, this process is also simple, but has an intersection instead of a union semantics. That is, the combination lists just those exception types, ET, such that (a) ET is mentioned in some `signals_only` clause and (b) for each `signals_only` clause, the exception ET is a subtype of some type mentioned in that clause. If this results in no exception types being listed in the result, the result uses the list `\nothing`. A concrete example of how the *signals-only-clause*s within a specification case are combined is shown in Figure 20, assuming that `StackException` is a subtype of `Exception`, and that `StackPushException` is a subtype of `StackException`.

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 governed by an `if` which is the disjunction of the original conditions.[3] For example, the *duration-clause*s within a specification case are combined as shown in Figure 21. (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 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 22. 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 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 23.

An example of this desugaring step is given in Figure 24.

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] Thanks to David Cok and Steve Edwards for suggesting that the entire expression be governed by this disjunction.

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

Figure 21: 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 22: Combining multiple *requires-clause*s; the combination of *ensures-clause*s, *diverges-clause*s, and *when-clause*s is similar.

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

Figure 23: Combining multiple *signals-clause*s.

```
/*@ public behavior
  @   assignable theElements;
  @   assignable size;
  @   duration 2*MC if MC>0;
  @   duration 50*CT;
  @   ensures \old(size >= 2);
  @   ensures size() == \old(size - 2);
  @   signals_only Exception;
  @   signals_only EmptyException,
  @                TooSmallException;
  @   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;
  @   duration Long.min(
  @       (MC>0 ? 2*MC, Long.MAX_VALUE),
  @       (true ? 50*CT, Long.MAX_VALUE))
  @     if ((MC>0) || true);
  @   ensures \old(size >= 2)
  @     && size() == \old(size - 2);
  @   signals_only EmptyException,
  @                TooSmallException;
  @   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 24: Example of the desugaring that eliminates multiple clauses of the same kind within a specification case.

```
/*@ public behavior
  @ {|
  @   requires elem >= 0;
  @   assignable size, elems[*];
  @   ensures \result == size
  @     && size == old_size + 1
  @     && ((0 <= i && i < old_size)
  @         ==> elems[i] == \old(elems[i]))
  @     && elems[size-1] == elem;
  @   signals_only \nothing;
  @  also
  @   requires elem < 0;
  @   assignable \nothing;
  @   ensures false;
  @   signals_only IllegalArgumentException;
  @ |}
  @*/
public Object push(int elem);
```

⇒

```
/*@ public behavior
  @ {|
  @   requires elem >= 0;
  @   assignable size, elems[*];
  @   ensures
  @     \only_assigned(size, elems[*])
  @     && (\result == size
  @     && size == old_size + 1
  @     && ((0 <= i && i < old_size)
  @         ==> elems[i] == \old(elems[i]))
  @     && elems[size-1] == elem);
  @   signals_only IllegalArugmentException;
  @   signals (Exception e) false;
  @  also
  @   requires elem < 0;
  @   assignable size, elems[*];
  @   ensures \only_assigned(\nothing)
  @     && (false);
  @   signals_only IllegalArgumentException;
  @ |}
  @*/
public Object push(int elem);
```

Figure 25: Example of making all *assignable-clause*s have the same frame, and all *signals-only-clause*s have the same list of exceptions.

3.10 Make Assignable and Signals_Only Clauses the Same

Before proceeding to the next step of combining *spec-case*s at each visibility level, we must have all *spec-case*s have the same *assignable-clause*s (frames) [20]. If this were not done, the semantics of the frames would not be preserved.

For very similar reasons, at this step we also make all the signals only clauses have the same list of exceptions that may be thrown.

An example of this step is given in Figure 25.

The general process is described in Figure 26. Let SR be the union of all the lists of *store-ref*s in the assignable clauses of the *generic-spec-case*s. (In taking unions, the *store-ref-list* `\everything` is the universal set, and `\nothing` is the empty set.) Then each assignable clause in each *generic-spec-case* is replaced by "`assignable` SR`;`". Furthermore, for each i the normal and exceptional postcondition in the i^{th} *generic-spec-case* has `\only_assigned(`SR_i`)` conjoined to it. This predicate records the original intent of the frame for that specification case.

In this figure, let ETL be the union of all the lists of *reference-type*s in the *generic-spec-case*s. Then each `signals_only` clause in each *generic-spec-case* is replaced by "`signals_only` ETL`;`". Furthermore, for each i the exceptional postcondition in the i^{th} *generic-spec-case* has `false` conjoined to it if the i^{th} *signals-only-clause* is "`signals_only \nothing;`" and otherwise, if the the i^{th} *signals-only-clause* is

`signals_only` $ET_{i,1}$, ..., $ET_{i,j}$`;`

then the i^{th} exceptional postcondition has the predicate we write as "e `instanceof` ETL_i," conjoined to it. This predicate stands for the disjunction

(e `instanceof` $ET_{i,1}$ `||` $\cdots$ `||` e `instanceof` $ET_{i,j}$)

This predicate records the original intent of the `signals_only` clause for that specification case.

```
V behavior
{|
    forall-var-decls_1
    old-var-decls_1
    requires P_1;
    diverges T_1;
    assignable SR_1;
    when W_1;
    working_space WSE_1 if W_1;
    duration DE_1 if D_1;
    ensures Q_1;
    signals_only ETL_1;
    signals (Exception e) EX_1;
also ... also
    forall-var-decls_k
    old-var-decls_k
    requires P_k;
    diverges T_k;
    assignable SR_k;
    when W_k;
    working_space WSE_k if W_k;
    duration DE_k if D_k;
    ensures Q_k;
    signals_only ETL_k;
    signals (Exception e) EX_k;
|}
```

⇒

```
V behavior
{|
    forall-var-decls_1
    old-var-decls_1
    requires P_1;
    diverges T_1;
    assignable SR;
    when W_1;
    working_space WSE_1 if W_1;
    duration DE_1 if D_1;
    ensures \only_assigned(SR_1) && (Q_1);
    signals_only ETL;
    signals (Exception e) \only_assigned(SR_1) && (e instanceof ETL_1) && (EX_1) ;
also ... also
    forall-var-decls_k
    old-var-decls_k
    requires P_k;
    diverges T_k;
    assignable SR;
    when W_k;
    working_space WSE_k if W_k;
    duration DE_k if D_k;
    ensures \only_assigned(SR_k) && (Q_k);
    signals_only ETL;
    signals (Exception e) \only_assigned(SR_k) && (e instanceof ETL_k) && (EX_k;
|}
```

Figure 26: Making *assignable-clause*s have the same frame. Here SR is the union of SR_1 through SR_k and ETL is the union of ETL_1 through ETL_k.

3.11 Desugaring Also Combinations

At this point, there is a effective single method specification, with at most one *spec-case* for each visibility level. Within each visibilty level, the *spec-case*s in each *generic-spec-case-seq* have the same frame and same `signals_only` clause. So now, 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 27. 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 [31, 15].

There is one additional detail that needs to be mentioned about Figure 27 — 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 28.

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 further 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 the conjunction rule illustrated in Section 3.9 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 [15], 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 [28, 29], and the conditions needed to check them also need to be formally stated (as has been done to some extent for Larch/C++ [14, 15]). 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 [25, 2].

Acknowledgments

Thanks to Albert Baker, Abhay Bhorkar, Michael Butler, Lilian Burdy, Yoonsik Cheon, Curtis Clifton, David R. Cok, Stephen Edwards, Michael D. Ernst, Bart Jacobs, Joe Kiniry, K. Rustan M. Leino, Peter Müller, Arnd Poetzsch-Heffter, Claude Marche, Erik Poll, Clyde Ruby, Edwin Rodríguez, Raymie Stata, Mark Utting, and Joachim van den Berg for many discussions about the syntax and semantics of such specifications. Thanks especially to David Cok, Erik Poll, and Edwin Rodríguez for discussions about the semantics of `pure`. Special thanks to David Cok for urging us to treat omitted specifications, `pure`, and `non_null`, and for insisting on understanding the interaction between these two, as well as for discussions about the details of these desugarings. Thanks to Yoonsik Cheon, Curtis Clifton, David Cok, Steve Edwards, Don Pigozzi, Clyde Ruby, Murali Sitaraman, and Wallapak Tavanapong for comments on earlier drafts.

A Definition of addHeaders and addBody

This section defines the auxiliary functions addHeaders and addBody, which are used to add preconditions and body clauses to specification cases.

The addHeaders function is fairly simple, and just has to distinguish the 4 different kinds of specification cases, as shown in Figure 29.

```
V behavior
{|
    forall-var-decls_1
    old-var-decls_1
    requires P_1;
    diverges T_1;
    assignable SR;
    when W_1;
    working_space WSE_1 if W_1;
    duration DE_1 if D_1;
    ensures Q_1;
    signals_only ETL;
    signals (Exception e) EX_1;
also ... also
    forall-var-decls_k
    old-var-decls_k
    requires P_k;
    diverges T_k;
    assignable SR;
    when W_k;
    working_space WSE_k if W_k;
    duration DE_k if D_k;
    ensures Q_k;
    signals_only ETL;
    signals (Exception e) EX_k;
|}
```

⇒

```
V behavior
   forall-var-decls_1 ... forall-var-decls_k
   old-var-decls_1 ... old-var-decls_k
   requires (P_1) || (P_2) ... || (P_k);
   diverges ((P_1) ==> T_1) && ... && ((P_k) ==> T_k);
   assignable SR;
   when (\old(P_1) ==> W_1) && ... && (\old(P_k) ==> W_k);
   working_space Integer.min((\old(P_1 && W_1) ? WSE_1 : Integer.MAX_VALUE),
                             ..., (\old(P_k && W_k) ? WSE_k : Integer.MAX_VALUE) ...)
               if ((P_1 && W_1) || ... || (P_k && W_k));
   duration Long.min((\old(P_1 && D_1) ? DE_1 : Long.MAX_VALUE),
                     ..., (\old(P_k && D_k) ? DE_k : Long.MAX_VALUE) ...)
               if ((P_1 && D_1) || ... || (P_k && D_k));
   ensures (\old(P_1) ==> Q_1) && ... && (\old(P_k) ==> Q_k);
   signals_only ETL;
   signals (Exception e) (\old(P_1) ==> EX_1) && ... && (\old(P_k) ==> EX_k);
```

Figure 27: 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 28: Example of the desugaring that eliminates `also`.

addHeaders : *spec-case* × *spec-header* → *spec-case*
addHeaders(*V* `behavior` *generic-spec-case*, *R*) =
    V `behavior` *R* `{|` *generic-spec-case* `|}`
addHeaders(*V* `exceptional_behavior` *exceptional-spec-case*, *R*) =
    V `exceptional_behavior` *R* `{|` *exceptional-spec-case* `|}`
addHeaders(*V* `normal_behavior` *normal-spec-case*, *R*) =
    V `normal_behavior` *R* `{|` *normal-spec-case* `|}`
addHeaders(*generic-spec-case*, *R*) =
    R `{|` *generic-spec-case* `|}`

Figure 29: Definition of addHeaders.

The addBody function is a bit more complex, as it has to recurse into the nested structure of specification cases to find places where a *simple-spec-body-clause* may occur. It is defined in Figures 30 and 31.

addBody : *spec-case* × *simple-spec-body-clause* → *spec-case*
addBody(V `behavior` *generic-spec-case*, C) = V `behavior` addBody(*generic-spec-case*, C)
addBody(V `exceptional_behavior` *exceptional-spec-case*, C) =
    if C is an *ensures-clause*
    then V `exceptional_behavior` *exceptional-spec-case*
    else V `exceptional_behavior` addBody(*exceptional-spec-case*, C)
addBody(V `normal_behavior` *normal-spec-case*, C) =
    if C is a *signals-clause*
    then V `normal_behavior` *normal-spec-case*
    else V `normal_behavior` addBody(*normal-spec-case*, C)
addBody(*generic-spec-case*, C) = addBody(*generic-spec-case*, C)

Figure 30: First part of the definition of addBody, which is overloaded for different kinds of specification case syntax.

addBody : *generic-spec-case* × *simple-spec-body-clause* → *generic-spec-case*
addBody(*spec-var-decls spec-header generic-spec-body*, C) =
    spec-var-decls spec-header addBody(*generic-spec-body*, C)
addBody(*spec-var-decls spec-header*, C) = *spec-var-decls spec-header* C
addBody(*spec-header*, C) = *spec-header* C
addBody(*spec-header generic-spec-body*, C) = *spec-header* addBody(*generic-spec-body*, C)
addBody(*spec-var-decls generic-spec-body*, C) = *spec-var-decls* addBody(*generic-spec-body*, C)
addBody(*generic-spec-body*, C) = addBody(*generic-spec-body*, C)

addBody : *generic-spec-body* × *simple-spec-body-clause* → *generic-spec-body*
addBody(*simple-spec-body*, C) = C *simple-spec-body*
addBody(`{|` *generic-spec-case*$_1$ ... *generic-spec-case*$_n$ `|}`, C) =
    `{|` addBody(*generic-spec-case*$_1$, C) ... addBody(*generic-spec-case*$_n$, C) `|}`

addBody : *normal-spec-case* × *simple-spec-body-clause* → *normal-spec-case*
addBody(*spec-var-decls spec-header normal-spec-body*, C) =
    spec-var-decls spec-header addBody(*normal-spec-body*, C)
addBody(*spec-var-decls spec-header*, C) = *spec-var-decls spec-header* C
addBody(*spec-header*, C) = *spec-header* C
addBody(*spec-header normal-spec-body*, C) = *spec-header* addBody(*normal-spec-body*, C)
addBody(*spec-var-decls normal-spec-body*, C) = *spec-var-decls* addBody(*normal-spec-body*, C)
addBody(*normal-spec-body*, C) = addBody(*normal-spec-body*, C)

addBody : *normal-spec-body* × *simple-spec-body-clause* → *normal-spec-body*
addBody(*normal-spec-clause*$_1$ ... *normal-spec-clause*$_n$, C) =
    C *normal-spec-clause*$_1$ ... *normal-spec-clause*$_n$
addBody(`{|` *normal-spec-case*$_1$ ... *normal-spec-case*$_n$ `|}`, C) =
    `{|` addBody(*normal-spec-case*$_1$, C) ... addBody(*normal-spec-case*$_n$, C) `|}`

addBody : *exceptional-spec-case* × *simple-spec-body-clause* → *exceptional-spec-case*
addBody(*spec-var-decls spec-header exceptional-spec-body*, C) =
    spec-var-decls spec-header addBody(*exceptional-spec-body*, C)
addBody(*spec-var-decls spec-header*, C) = *spec-var-decls spec-header* C
addBody(*spec-header*, C) = *spec-header* C
addBody(*spec-header exceptional-spec-body*, C) = *spec-header* addBody(*exceptional-spec-body*, C)
addBody(*spec-var-decls exceptional-spec-body*, C) = *spec-var-decls* addBody(*exceptional-spec-body*, C)
addBody(*exceptional-spec-body*, C) = addBody(*exceptional-spec-body*, C)

addBody : *exceptional-spec-body* × *simple-spec-body-clause* → *exceptional-spec-body*
addBody(*exceptional-spec-clause*$_1$ ... *exceptional-spec-clause*$_n$, C) =
    C *exceptional-spec-clause*$_1$ ... *exceptional-spec-clause*$_n$
addBody(`{|` *exceptional-spec-case*$_1$ ... *exceptional-spec-case*$_n$ `|}`, C) =
    `{|` addBody(*exceptional-spec-case*$_1$, C) ... addBody(*exceptional-spec-case*$_n$, C) `|}`

Figure 31: Second part of the definition of addBody.

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. *International Journal on Software Tools for Technology Transfer (STTT)*, 7, February 2005. To appear.

[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.

[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] David R. Cok and Joseph Kiniry. ESC/Java2: Uniting ESC/Java and JML. Technical report, University of Nijmegen, 2004. NIII Technical Report NIII-R0413.

[9] 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.

[10] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 234–245, New York, June 17–19 2002. ACM Press.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Iowa State University, Department of Computer Science, April 2005. See `www.jmlspecs.org`.

[17] 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.

[18] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, and Joseph Kiniry. JML reference manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, April 2005.

[19] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 144–153. ACM, October 1998.

[20] K. Rustan M. Leino and Rajit Manohar. Joining specification statements. *Theoretical Computer Science*, 216(1-2):375–394, March 1999.

[21] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical note, Compaq Systems Research Center, October 2000.

[22] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)*, volume 37, 5 of *SIGPLAN*, pages 246–257, New York, June 17–19 2002. ACM Press.

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

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

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

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

[27] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. Technical Report SAnToS-TR2004-10, Kansas State University, Department of Computing and Information Sciences, May 2005. To appear in *ECOOP 2005*.

[28] 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.

[29] 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.

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

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