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


Copyright © 2000, 2001, 2003 by Iowa State University.
This document is distributed under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2, or (at your option) any later version.

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
arunragnar@microsoft.com, leavens@cs.iastate.edu

May 20, 2003

Abstract

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

1 Introduction

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

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

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

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

2 JML Method Specifications

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

∗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 and CCR-0113181. This paper is a minor adaptation of chapter 2 of Raghavan’s Master’s thesis [21], which itself was adapted from a previous version of this document.

1 The JML release is available from http://www.jmlspecs.org
2.1 Basic Method Specifications

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

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

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

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

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

The second clause in Figure 1 is an old variable declaration. This is used to abbreviate side-effect free expressions that can be used throughout the rest of the specification. The values of such expressions are evaluated in the pre-state of the method’s execution; that is, the variables can be thought of as initialized just after parameter passing to the values of the expressions. This timing is the explanation for the keyword “old”. Like the variables declared in the forall clause, the variables declared in the old clause are logical variables, and have a scope that extends for the rest of this specification case. In the example, \( \text{old.size} \) 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 \( \text{old()} \) expression; the semantic connection is that \( \text{old()} \) expressions can be given a semantics using \( \text{old} \) declarations.

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

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

The assignable clause gives a frame axiom [2] for a specification case. The frame axiom lists the locations that the method may assign to, and under what conditions it may assign to them. Given a specific pre-state, only the locations mentioned whose conditions are true in that pre-state, or that are not depended on by such locations [20], may be assigned to by the method’s execution, all other
public class NonNegStack {

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

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

Figure 1: An example of the core features of JML method specification cases is given in the specification of the push method, which appears just above the code for push.
locations may not be assigned to in such an execution of the method. (Note that this does not say that the method must assign to these locations it is permitted to assign to, only that it may do so.) If omitted in a heavyweight specification case, there is no restriction on what locations the method may assign to, so it is usually important to give the frame axiom for a method.

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

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

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

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

The **ensures** clause gives a normal postcondition for a method. The JML semantics is that, when the method returns normally, the relationship between the pre-state and the post-state of the method must satisfy the normal postcondition given in the **ensures** clause. In the postcondition for a method with a non-void return type, the notation \( \text{result} \) 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 \( \text{elem} \) was not negative, and each element of the \( \text{elems} \) that was previously defined has the same value, and the new element added is \( \text{elem} \), and the result is the post-state value of the stack’s size. Since the argument \( \text{elem} \) must be positive for the normal postcondition to be true, the method may not return normally when \( \text{elem} \) is negative.

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

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

- terminate normally, in which case the relation between the pre-state and the post-state must satisfy the normal postcondition given by the **ensures** clause,
• throw an exception of type $ET$, in which case the relation between the pre-state and the post-state must satisfy the exceptional postcondition in all signals clauses that name a supertype of $ET$ (including $ET$ itself),

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

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

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

2.2 Extending Specifications

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

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

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

2.3 JML Grammar

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

- Nonterminal symbols are written in italics.
- Terminal symbols appear in teletype font
- Symbols within square brackets ([ ]) are optional.
- An ellipses (...) indicates that the preceding nonterminal or group of optional text can be repeated zero or more times.

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

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

generic-spec-case ::= [ spec-var-decls ] spec-header [ generic-spec-body ]
   | [ spec-var-decls ] generic-spec-body
spec-header ::= requires-clause | requires-clause | . . .
generic-spec-body ::= simple-spec-body | '{1' generic-spec-case-seq '}'
generic-spec-case-seq ::= generic-spec-case [ also generic-spec-case ] . . .
simple-spec-body ::= simple-spec-body-clause | simple-spec-body-clause | . . .
simple-spec-body-clause ::= diverges-clause
   | assignable-clause | accessible-clause
   | when-clause | working-space-clause
   | duration-clause | ensures-clause | signals-clause

Figure 3: Generic specification cases, used for “lightweight” specification cases.

behavior-spec ::= [ privacy ] behavior generic-spec-case
   | [ privacy ] exceptional_behavior exceptional-spec-case
   | [ privacy ] normal_behavior normal-spec-case
exceptional-spec-case ::= [ spec-var-decls ] spec-header [ exceptional-spec-body ]
   | [ spec-var-decls ] exceptional-spec-body
privacy ::= public | protected | private
exceptional-spec-body ::= exceptional-spec-clause | exceptional-spec-clause | . . .
   | '{1' exceptional-spec-case-seq '}'
exceptional-spec-clause ::= diverges-clause
   | assignable-clause | accessible-clause
   | when-clause | working-space-clause
   | duration-clause | signals-clause
exceptional-spec-case-seq ::= exceptional-spec-case [ also exceptional-spec-case ] . . .
normal-spec-case ::= [ spec-var-decls ] spec-header [ normal-spec-body ]
   | [ spec-var-decls ] normal-spec-body
normal-spec-body ::= normal-spec-clause | normal-spec-clause | . . .
   | '{1' normal-spec-case-seq '}'
normal-spec-clause ::= diverges-clause
   | assignable-clause | accessible-clause
   | when-clause | working-space-clause
   | duration-clause | ensures-clause
normal-spec-case-seq ::= normal-spec-case [ also normal-spec-case ] . . .

Figure 4: Behavior specification syntax, used for “heavyweight” specification cases.
3 Desugaring

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

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

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

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

3. Combine each extending-specification (in a subclass or refinement) with the inherited or refined method-specification 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-clauses (see Section 3.5).

6. Desugar the also combinations within the spec-cases. (see Section 3.6).

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

3.1 Desugaring Nested Specifications

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

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

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

The same kind of desugaring applies to a generic-spec-case and to an exceptional-spec-case.
/*@ requires x > 0; @
@ requires x % 2 == 1; @
@ ensures \result == 3*x + 1; @
@ also @
@ requires x > 0; @
@ 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.

forall quantified-var-decl0; ... forall quantified-var-decl0,fo;
old old-var-decl0,1; ... old old-var-decl0,vo;
requires P0,1; ... requires P0,ro;
{}
forall quantified-var-decl1,1; ... forall quantified-var-decl1,f1;
old old-var-decl1,1; ... old old-var-decl1,v1;
requires P1,1; ... requires P1,ro1;
normal-spec-case-body1
also ...
also
forall quantified-var-declk,1; ... forall quantified-var-declk,fk;
old old-var-declk,1; ... old old-var-declk,vk;
requires Pk,1; ... requires Pk,rok;
normal-spec-case-bodyk
}
}

⇒
forall quantified-var-decl0,1; ... forall quantified-var-decl0,fo;
forall quantified-var-decl1,1; ... forall quantified-var-decl1,f1;
old old-var-decl0,1; ... old old-var-decl0,vo
old old-var-decl1,1; ... old old-var-decl1,v1
requires P0,1; ... requires P0,ro0;
requires P1,1; ... requires P1,ro1;
normal-spec-case-body1
also ...
also
forall quantified-var-decl0,1; ... forall quantified-var-decl0,fo;
forall quantified-var-declk,1; ... forall quantified-var-declk,fk;
old old-var-decl0,1; ... old old-var-decl0,vo
old old-var-declk,1; ... old old-var-declk,vk
requires P0,1; ... requires P0,ro0;
requires Pk,1; ... requires Pk,rok;
normal-spec-case-bodyk

Figure 7: Desugaring nested specifications in a normal-spec-case.
3.2 Desugaring Normal and Exceptional Specifications

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

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

\[
\text{signals (Exception e) false;}
\]

which says that no exceptions can be thrown. Similarly, we desugar spec-cases 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-cases that are behavior-specs that use the keyword “behavior”. (It is not necessary for the desugaring to fill in the defaults for omitted clauses in such heavyweight specification cases [14 Appendix A], although this could be done as well.) However, these spec-cases may have different visibility, and for later processing it is convenient to have the specification cases grouped by visibility level. So, we combine spec-cases with the same visibility into a single behavior-spec, each of these having a generic-spec-case-seq, which contains each of the bodies of the behavior-specs 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-specs with the form described above, one for each privacy level.

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

3.3 Desugaring Inheritance and Refinement

Extending specifications augment an inherited specification from the specification of the same Java method in the superclass (if any), superinterfaces, and the refined specification (if any). As described in the previous step, the inherited specification has up to four spec-cases, 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-cases, one for each visibility level, again joined together with “also”. 
Inherited specification:

\[
\begin{align*}
V & \text{ behavior} \\
& \{ \\
& \quad \text{spec-case}_{A_1} \\
& \quad \text{also ... also} \\
& \quad \text{spec-case}_{A_k} \\
& \} \\
\end{align*}
\]

Combined specification:

\[
\begin{align*}
V & \text{ behavior} \\
& \{ \\
& \quad \text{spec-case}_{A_1} \\
& \quad \text{also ... also} \\
& \quad \text{spec-case}_{A_k} \\
& \quad \text{also} \\
& \quad \text{spec-case}_{B_1} \\
& \quad \text{also ... also} \\
& \quad \text{spec-case}_{B_n} \\
& \} \\
\end{align*}
\]

Extending specification:

\[
\begin{align*}
& \text{also} \\
& V \text{ behavior} \\
& \{ \\
& \quad \text{spec-case}_{B_1} \\
& \quad \text{also ... also} \\
& \quad \text{spec-case}_{B_n} \\
& \} \\
\end{align*}
\]

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

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

An extending specification starts with "\textit{also}" and may have \textit{spec-cases} with each of the four visibilities. For each visibility level, \(V\), we combine the inherited specification with the corresponding \textit{spec-case} of the extending specification, as shown in Figure 10.
3.4 Standardizing Signals Clauses

At this point, there may be several *signals-clauses* 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-clauses*, 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-clauses* 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:

\[
\text{signals (}\text{ET n) } P; \Rightarrow \text{signals (Exception e) (e instanceof ET) } \Rightarrow \left(\left(\text{((ET)e)}/n\right)P\right);
\]

where the notation \(\left(\text{((ET)e)}/n\right)P\) means \(P\) with the cast expression \(((\text{ET}e))\) substituted for free occurrences of \(n\).

An example of this desugaring is shown in Figure 11.

![Figure 11: Example of the desugaring that standardizes signals-clauses.](image)

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-clauses*, and all of the clauses that can occur in a *simple-spec-body-clause* (see Figure 3).

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

For the *working-space-clauses* and *duration-clauses*, 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. For example, the *duration-clauses* within a specification case are combined as shown in Figure 13. (Again, if the “if” and the following predicate are omitted after a *spec-expression*, “if true” is used as a default.)

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

---

2 Thanks to David Cok and Steve Edwards for this suggestion.
assignable $SR_{1,1}$ if $M_{1,1}$, ..., $SR_{1,n}$ if $M_{1,n}$;
...  
assignable $SR_{k,1}$ if $M_{k,1}$, ..., $SR_{k,m}$ if $M_{k,m}$;
⇒
assignable $SR_{1,1}$ if $M_{1,1}$, ..., $SR_{1,n}$ if $M_{1,n}$, ..., $SR_{k,1}$ if $M_{k,1}$, ..., $SR_{k,m}$ if $M_{k,m}$;

Figure 12: Combining multiple assignable-clauses; the combination of accessible-clauses is similar.

---

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

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

---

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

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

---

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

Figure 15: Combining multiple signals-clauses.
Figure 16: Example of the desugaring that eliminates multiple clauses of the same kind within a specification case.

**3.6 Desugaring Also Combinations**

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

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

An example of this desugaring is given in Figure 18.

**4 Conclusion**

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

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

\begin{verbatim}
forall-var-decls1
old-var-decls1
requires \( P_1 \);
diverges \( T_1 \);
assignable \( WSR_{1,1} \) if \( M_{1,1} \), ...., \( WSR_{1,m_1} \) if \( M_{1,m_1} \);
accessible \( RSR_{1,1} \) if \( N_{1,1} \), ...., \( RSR_{1,m_1} \) if \( N_{1,m_1} \);
when \( W_1 \);
working_space \( WSE_1 \) if \( W_1 \);
duration \( DE_1 \) if \( D_1 \);
ensures \( Q_1 \);
signals (Exception e) \( EX_1 \);
\end{verbatim}

also ... also

\begin{verbatim}
forall-var-decls_k
old-var-decls_k
requires \( P_k \);
diverges \( T_k \);
assignable \( WSR_{k,1} \) if \( M_{k,1} \), ...., \( WSR_{k,m_k} \) if \( M_{k,m_k} \);
accessible \( RSR_{k,1} \) if \( N_{k,1} \), ...., \( RSR_{k,m_k} \) if \( N_{k,m_k} \);
when \( W_k \);
working_space \( WSE_k \) if \( W_k \);
duration \( DE_k \) if \( D_k \);
ensures \( Q_k \);
signals (Exception e) \( EX_k \);
\end{verbatim}

\Rightarrow

\textbf{V} behavior

\begin{verbatim}
forall-var-decls_1 \ldots \ forall-var-decls_k
old-var-decls_1 \ldots old-var-decls_k
requires \( P_1 \ || \ldots \ || \ P_k \);
diverges \((P_1) \implies (T_1) \land \ldots \land ((P_k) \implies (T_k))\);
assignable \( WSR_{1,1} \) if \( P_1 \land M_{1,1} \), ...., \( WSR_{1,m_1} \) if \( P_1 \land M_{1,m_1} \);
\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \\
accessible \( RSR_{1,1} \) if \( P_1 \land N_{1,1} \), ...., \( RSR_{1,m_1} \) if \( P_1 \land N_{1,m_1} \);
\ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \ldots \\
when \( \text{\texttt{\langle\texttt{old}(P_1) \implies W_1\rangle \land \ldots \land \langle\texttt{old}(P_k) \implies W_k\rangle}} \);
working_space \text{\texttt{\langle\texttt{old}(P_1 \land W_1) \land \ldots \land \text{\texttt{\langle\texttt{old}(P_k \land W_k) \land \ldots \land \text{\texttt{\langle\texttt{old}(Q_1 \land \ldots \land \text{\texttt{\langle\texttt{old}(Q_k \land \ldots \land \text{\texttt{\langle\texttt{old}(EX_1 \land \ldots \land EX_k)\rangle}}}))}}))}}))};
normalsize
\end{verbatim}

\begin{verbatim}
\text{\texttt{\langle\texttt{old}(P_1 \land D_1) \land \ldots \land \text{\texttt{\langle\texttt{old}(P_k \land D_k) \land \ldots \land \text{\texttt{\langle\texttt{old}(Q_1 \land \ldots \land \text{\texttt{\langle\texttt{old}(Q_k \land \ldots \land \text{\texttt{\langle\texttt{old}(EX_1 \land \ldots \land EX_k)\rangle}}}))}}))};
\end{verbatim}

ensures \( \text{\texttt{\langle\texttt{old}(P_1) \implies Q_1 \land \ldots \land \langle\texttt{old}(P_k) \implies Q_k \rangle}});\)
signals (Exception e) \( \text{\texttt{\langle\texttt{old}(P_1) \implies EX_1 \land \ldots \land \langle\texttt{old}(P_k) \implies EX_k \rangle}});\)

\end{verbatim}

Figure 17: Desugaring \textit{also} combinations
the conjunction rule illustrated in Section 3.5 at any time. Allowing different orderings brings up the possibility that applying the desugaring steps in different orders might produce different results; we should try to prove that this cannot happen.

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

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

Acknowledgements

Thanks to Albert Baker, Abhay Bhorkar, Yoonsik Cheon, Curtis Clifton, Stephen Edwards, Bart Jacobs, K. Rustan M. Leino, Peter Müller, Arnd Poetzsch-Heffter, Clyde Ruby, Raymie Stata, and Joachim van den Berg for many discussions about the syntax and semantics of such specifications. Thanks to Yoonsik Cheon, Curtis Clifton, David Cok, Don Pigozzi, Clyde Ruby, Murali Sitaraman, and Wallapak Tavanapong for comments on earlier drafts.

References


Figure 18: Example of the desugaring that eliminates also.


