Java Modeling Language Reference Manual

David R. Cok, Gary T. Leavens, Mattias Ulbrich, TBD

DRAFT March 26, 2018

Contents

1 Introduction **3**
- 1.1 Behavioral Interface Specifications . . . 3
- 1.2 A First Example . . . 4
- 1.3 What is JML Good For? . . . 4
- 1.4 Status, Plans and Tools for JML . . . 4
- 1.5 Purpose of this document . . . 5
- 1.6 Previous JML reference manual . . . 5
- 1.7 Historical Precedents and Antecedents . . . 5
- 1.8 Acknowledgments . . . 7

2 Structure of this Manual **9**
- 2.1 Typographical conventions . . . 9
- 2.2 Grammar . . . 9

3 JML concepts **11**
- 3.1 JML and Java compilation units . . . 12
- 3.2 `.jml` files . . . 12
  - 3.2.1 Combining Java and JML files . . . 14
- 3.3 Specification inheritance . . . 15
- 3.4 JML modifiers and Java annotations . . . 16
  - 3.4.1 Modifiers . . . 16
  - 3.4.2 Type modifiers . . . 17
- 3.5 Model and Ghost . . . 17
- 3.6 Visibility . . . 17
- 3.7 Evaluation and well-formedness of JML expressions . . . 17
- 3.8 Null and non-null references . . . 17
- 3.9 Static and Instance . . . 18
- 3.10 Observable purity . . . 18
- 3.11 Location sets and Dynamic Frames . . . 18
- 3.12 Arithmetic modes . . . 18
- 3.13 Immutable types and functions . . . 18
- 3.14 Race condition detection . . . 18
- 3.15 Redundant specifications . . . 18
- 3.16 Controlling warnings . . . 18

3.17 org.jmlspecs.lang package 19
3.18 Interaction with other tools 19
3.18.1 Interaction with Type Annotations in Java 1.8 19
3.18.2 Interaction with the Checker framework 19
3.18.3 Interaction with FindBugs 19
3.19 Core JML 19

4 JML Syntax **24**
4.1 Textual form of JML specifications 24
4.2 JML Syntax 26
4.2.1 Syntax of JML specifications 26
4.2.2 Conditional JML specifications 26

5 JML Types **28**
5.1 `\bigint` 28
5.2 `\real` 29
5.3 `\TYPE` 29

6 JML Specifications for Packages and Compilation Units **30**
6.1 Model import statements 30
6.2 Default imports 31
6.3 Issues with model import statements 31

7 Specifications for Java types in JML **33**
7.1 Modifiers for type declarations 33
7.1.1 non_null_by_default, nullable_by_default, @NonNullByDefault, @NullableByDefault 34
7.1.2 pure and @Pure 34
7.1.3 @Options 34
7.2 invariant clause 34
7.3 constraint clause 35
7.4 initially clause 35
7.5 ghost fields 35
7.6 model fields 35
7.7 represents clause 35
7.8 model methods and model classes 35
7.9 initializer and static_initializer 35
7.10 axiom 36
7.11 readable if clause and writable if clause 36
7.12 monitors_for clause 36

8 JML Method specifications **37**
8.1 Structure of JML method specifications 37
8.1.1 Behaviors 39
8.1.2 Nested specification clauses 39
8.1.3 Specification inheritance and the `code` modifier 40

8.1.4 Visibility . . . 40
8.1.5 Grammar of method specifications . . . 41
8.2 Method specifications as Annotations . . . 42
8.3 Modifiers for methods . . . 42
8.4 Common JML method specification clauses . . . 42
8.4.1 `requires` clause . . . 42
8.4.2 `ensures` clause . . . 42
8.4.3 `assignable` clause . . . 42
8.4.4 `signals` clause . . . 42
8.4.5 `signals_only` clause . . . 42
8.5 Advanced JML method specification clauses . . . 43
8.5.1 `accessible` clause . . . 43
8.5.2 `diverges` clause . . . 43
8.5.3 `measured_by` clause . . . 43
8.5.4 `when` clause . . . 43
8.5.5 `old` clause . . . 43
8.5.6 `forall` clause . . . 43
8.5.7 `duration` clause . . . 43
8.5.8 `working_space` clause . . . 43
8.5.9 `callable` clause . . . 44
8.5.10 `captures` clause . . . 44
8.6 Model Programs (`model_program` clause) . . . 44
8.6.1 Structure and purpose of model programs . . . 44
8.6.2 `extract` clause . . . 44
8.6.3 `choose` clause . . . 44
8.6.4 `choose_if` clause . . . 44
8.6.5 `or` clause . . . 44
8.6.6 `returns` clause . . . 44
8.6.7 `continues` clause . . . 44
8.6.8 `breaks` clause . . . 45
8.7 Modifiers for method specifications . . . 45
8.7.1 pure and @Pure . . . 45
8.7.2 non_null, nullable, @NonNull, and @Nullable . . . 45
8.7.3 model and @Model . . . 45
8.7.4 spec_public, spec_protected, @SpecPublic, and @SpecProtected 45
8.7.5 helper and @Helper . . . 45
8.7.6 function and @Function . . . 45
8.7.7 query, secret, @Query, and @Secret . . . 45
8.7.8 code_java_math, code_bigint_math, code_safe_math . . . 46
8.7.9 skip_esc, skip_rac, @SkipEsc, and SkipRac . . . 46
8.7.10 options and @Options . . . 46
8.7.11 extract and @Extract . . . 46
8.8 TODO Somewhere . . . 46

9 Field Specifications **47**
9.1 Field and Variable Modifiers . . . 47

9.1.1 non_null and nullable (@NonNull, @Nullable) 47
9.1.2 spec_public and spec_protected (@SpecPublic, @SpecProtected) 47
9.1.3 `ghost` and `@Ghost` 49
9.1.4 `model` and `@Model` 49
9.1.5 `uninitialized` and `@Uninitialized` 49
9.1.6 `instance` and `@Instance` 49
9.1.7 `monitored` and `@Monitored` 49
9.1.8 `query`, `secret` and `@Query`, `@Secret` 49
9.1.9 `peer`, `rep`, `readonly` (`@Peer`, `@Rep`, `@Readonly`) 49
9.2 Datagroups: `in` and `maps` clauses 49
9.3 Ghost fields 49
9.4 Model fields 49

10 JML Statement Specifications **50**
10.1 `assert` statement and Java assert statement 50
10.2 `assume` statement 51
10.3 ghost and model declarations 51
10.4 `unreachable` statement 51
10.5 `reachable` statement 52
10.6 `set` and `debug` statements 53
10.7 loop specifications 53
10.8 statement (block) specification 54
10.9 `hence_by` statement 55

11 JML Expressions **56**
11.1 Syntax 57
11.2 Purity (no side-effects) 57
11.3 Java operations used in JML 57
11.4 Precedence of infix operations 57
11.5 Well-defined expressions 59
11.6 org.jmlspecs.lang.JML 60
11.7 Implies and reverse implies: `==>` `<==` 60
11.8 Equivalence and inequivalence: `<==>` `<=!=>` 61
11.9 JML subtype: `<:` 61
11.10 Lock ordering: `<#` `<#=` 62
11.11 `\result` 62
11.12 `\exception` 63
11.13 `\old`, `\pre`, and `\past` 63
11.14 `\key` 64
11.15 `\lblpos`, `\lblneg`, `\lbl`, and `JML.lblpos()`, `JML.lblneg()`, `JML.lbl()` 64
11.16 `\nonnullelements` 65
11.17 `\fresh` 66
11.18 informal expression: `(*...*)` and `JML.informal()` 67
11.19 `\type` 68
11.20 `\typeof` 68

11.21 \elemtype . . . 69
11.22 \is_initialized . . . 69
11.23 \invariant_for . . . 70
11.24 \not_modified . . . 71
11.25 \lockset and \max . . . 71
11.26 \reach . . . 71
11.27 \duration . . . 72
11.28 \working_space . . . 72
11.29 \space . . . 73

12 Arithmetic modes **74**
12.1 Description of arithmetic modes . . . 74
12.2 Semantics of Java math mode . . . 75
12.3 Semantics of Safe math mode . . . 76
12.4 Semantics of Bigint math mode . . . 77
12.5 Real arithmetic and non-finite values . . . 77

13 Universe types **78**

14 Model Programs **79**

15 Obsolete and Deprecated Syntax **80**

A Summary of Modifiers **81**

B Grammar Summary **84**

C Type Checking Summary **85**

D Verification Logic Summary **86**

E Differences in JML among tools **87**

F Misc stuff to move, incorporate or delete **88**
F.0.1 Finding specification files and the refine statement . . . 88
F.0.2 Model import statements . . . 89
F.0.3 Modifiers . . . 89
F.0.4 JML expressions . . . 91
F.0.5 Code contracts . . . 91
F.1 JML modifiers and Java annotations . . . 91
F.2 org.jmlspecs.lang package . . . 93
F.3 JML Syntax . . . 94
F.3.1 Syntax of JML specifications . . . 94
F.3.2 Conditional JML specifications . . . 94
F.3.3 Finding specification files and the refine statement . . . 95
F.3.4 Model import statements . . . 96
F.3.5 Modifiers . . . 97

F.3.6 JML expressions 98
F.3.7 JML types 98

G Statement translations **99**
G.1 While loop 99

H Java expression translations **101**
H.1 Implicit or explicit arithmetic conversions 101
H.2 Arithmetic expressions 101
H.3 Bit-shift expressions 102
H.4 Relational expressions 102
H.5 Logical expressions 102

General notes on things not to forget:

- enum types

- default specs for binary classes

- datagroups, JML.* utility functions, @Requires-style annotations. arithmetic modes, universe types

- interaction with JSR 308

- various @NonNull annotations in different packages

- visibility in JML

– Sorted First-order-logic

– individual subexpressions; optional expression form; optimization; usefulness for tracing

– RAC vs. ESC

– nomenclature

– lambda expressions

– other Java 6,7,8 features

– Specification of subtypes - cf Clyde Ruby's thesis

Preface

This document gives the definition of the Java Modeling Language (JML), a language in which to write behavioral specifications for Java programs. Since about 2000 or so, JML has been the de facto language for writing behavioral specifications for Java programs. It was first a vehicle for discussing theoretical and soundness issues in specification and verification of software. It then also became the language to use in education about verification, since Java was a commonly taught language in undergraduate curricula; it is also frequently a basis for master's and Ph.D. theses, sometimes not even known to the core JML researchers until questions arise. Finally, JML is now being used to help verify, or at least increase confidence in, critical industrial software.

With this broadening of the scope of JML, the JML community, and in particular the participants in the more-or-less annual JML workshops, considered that the long-standing and evolving Draft JML Reference manual [**?**] should be rewritten and expanded to represent the current state of JML. In the process, many outstanding semantic and syntactic issues have been either resolved or clearly stated. This document is the result of that collaborative effort. Consequently this document is a completely revised, rewritten and expanded reference maual for JML, while borrowing (and copying) much from the original document.

The document does not do some other things in which the reader may be interested:

- This document does not describe tools that implement JML or how to use those tools. Some such tools are
  - OpenJML — `www.openjml.org` — and its user guide: *TBD*
  - the KeY tool — `https://www.key-project.org/` — including a book about KeY: `https://www.key-project.org/thebook2/`
- This document is not a tutorial about writing specifications in JML. For such a tutorial, see *TBD* .

Chapter 1

Introduction

1.1 Behavioral Interface Specifications

JML is a *behavioral interface specification language* (BISL) that builds on the Larch approach [**?**] [**?**] and that found in APP [**?**] and Eiffel [**?**] [**?**]. In this style of specification, which might be called model-oriented [**?**], one specifies both the interface of a method or abstract data type and its behavior [**?**]. In particular JML builds on the work done by Leavens and others in Larch/C++ [**?**] [**?**] [**?**]. (Indeed, large parts of this manual are adapted wholesale from the Larch/C++ reference manual [**?**].) Much of JML's design was heavily influenced by the work of Leino and his collaborators [**?**] [**?**] [**?**] [**?**] [**?**] and then subsequently by Cok's work on ESC/Java2 [**?**] and OpenJML [**?**]. JML continues to be influenced by ongoing work in formal specification and verification. A collection of papers relating directly to JML and its design is found at `http://www.jmlspecs.org/papers.shtml`.

The *interface* of the method or type is the information needed to use it from other parts of a program. In the case of JML, this is the Java syntax and type information needed to call a method or use a field or type. For a method the interface includes such things as the name of the method, its modifiers (including its visibility and whether it is final) its number of arguments, its return type, what exceptions it may throw, and so on. For a field the interface includes its name and type, and its modifiers. For a type, the interface includes its name, its modifiers, its package, whether it is a class or interface, its supertypes, and the interfaces of the fields and methods it declares and inherits. JML specifies all such interface information using Java's syntax.

A *behavior* of a method or type describes a set of state transformations that it can perform. A behavior of a method is specified by describing

- a set of states in which calling the method is defined,
- a set of locations that the method is allowed to assign to (and hence change),

- and the relations between the calling state and the state in which it either returns normally, throws an exception, or for which it might not return to the caller.

The states for which the method is defined are formally described by a logical assertion, called the method's *precondition*. The allowed relationships between these states and the states that may result from normal return are formally described by another logical assertion called the method's *normal postcondition*. Similarly the relationships between these pre-states and the states that may result from throwing an exception are described by the method's *exceptional postcondition*. The states for which the method need not return to the caller are described by the method's *divergence condition*; however, explicit specification of divergence is rarely used in JML. The set of locations the method is allowed to assign to is described by the method's *frame condition* [**?**]. In JML one can also specify other aspects of behavior, such as the time a method can use to execute and the space it may need.

The behavior of an abstract data type (ADT), which is implemented by a class in Java, is specified by describing a set of abstract fields for its objects and by specifying the behavior of its methods (as described above). The abstract fields for an object can be specified either by using JML's model and ghost fields [**?**], which are specification-only fields, or by specifying some of the fields used in the implementation as `spec_public` or `spec_protected`. These declarations allow the specifier using JML to model an instance as a collection of abstract instance variables, in much the same way as other specification languages, such as Z [**?**] [**?**] or Fresco [**?**].

To write - take from DRM?

1.2 A First Example

To write - take from DRM?

Does the formal standard document require such an example? Would a reference to a good JML tutorial not be better?

1.3 What is JML Good For?

To write - take from DRM?

1.4 Status, Plans and Tools for JML

To write - take from DRM? Include somewhere a discussion of available tools and other resources, to include at least OpenJML, Key, OpenJML tutorial, Key book, etc.

1.5 Purpose of this document

The purpose of this standardization document is to define the syntax and formal semantics of JML as a language. The document also distinguishes core aspects of JML, which over the years have proved to be the most used and most important specification elements.

While the standard means to set a fixed meaning, JML is also a research vehicle. Consequently the community's understanding of the best semantics and of best practices evolves with further research. Thus the standard intends to be a common basis for tools and for discussion but does not mean to inhibit experimentation and proposals for change. Therefore we present a semantical framework in which new tools and approaches can be defined such that a deviation of the semantics from the standard can be clearly stated.

The purpose of JML is the specification of Java code in various contexts, ranging from dealing with isolated research challenges to covering industrial application cases.Moreover, JML specifications are to be interpreted by tools covering a wide variety of applications: from formal documentation, runtime assertion checking to full deductive verification. To make JML a versatile specification vehicle, the meaning of its annotations must be unambiguously clear. And *if* tools interpret a few language constructs differently, these differences must be easily and concise stated.

1.6 Previous JML reference manual

This reference manual builds on the previous draft JML Reference Manual [1], which has been evolving over many years and has many contributors. This current edition of the reference manual is largely a rewrite of the previous draft; it incorporates the experience of building tools for JML by the OpenJML and Key developers, many decisions about new features or deprecated features made at JML workshops, and discussions about JML on the JML mailing lists. This edition of the reference manual includes features that are proposed enhancements or clarifications of the consensus language definition. It also includes rationale for non-obvious language features and discussion of points that are under current debate or require extended explanation.

JML changes as the current version of Java changes. As this document is being written, Java 8 is the current version of Java, with Java 9 on the horizon. The version of JML presented here corresponds to Java 8. Java 8 has affected JML by, for example, the introduction of type annotations and lambda expressions.

1.7 Historical Precedents and Antecedents

To write - take from DRM? INclude description of other languages that have been

inspired by JML for other source languages

JML combines ideas from Eiffel [**?**] [**?**] [**?**] with ideas from model-based specification languages such as VDM [**?**] and the Larch family [**?**] [**?**] [**?**] [**?**]. It also adds some ideas from the refinement calculus [**?**] [**?**] [**?**] [**?**] [**?**]. In this section we describe the advantages and disadvantages of these approaches. Readers not interested in these historical precedents may want to skip this section.

Formal, model-based languages such as those typified by the Larch family build on ideas found originally in Hoare's work. Hoare used pre- and postconditions to describe the semantics of computer programs in his famous article [**?**]. Later Hoare adapted these axiomatic techniques to the specification and correctness proofs of abstract data types [**?**]. To specify an ADT, Hoare described a mathematical set of abstract values for the type, and then specified pre- and postconditions for each of the operations of the type in terms of how the abstract values of objects were affected. For example, one might specify a class `IntHeap` using abstract values of the form `empty` and `add(i,h)`, where `i` is an `int` and `h` is an `IntHeap`. These notations form a mathematical vocabulary used in the rest of the specification.

There are two advantages to writing specifications with a mathematically-defined abstract values instead of directly using Java variables and data structures. The first is that by using abstract values, the specification does not have to be changed when the particular data structure used in the program is changed. This permits different implementations of the same specification to use different data structures. Therefore the specification forms a contract between the rest of the program and the implementation, which ensures that the rest of the program is also independent of the particular data structures used [**?**] [**?**] [**?**] [**?**]. Second, it allows the specification to be written even when there are no implementation data structures, as is the case for `IntHeap`.

This idea of model-oriented specification has been followed in VDM [**?**], VDM-SL [**?**] [**?**], Z [**?**] [**?**], and the Larch family [**?**]. In the Larch approach, the essential elaboration of Hoare's original idea is that the abstract values also come with a set of operations. The operations on abstract values are used to precisely describe the set of abstract values and to make it possible to abbreviate interface specifications (i.e., pre- and postconditions for methods). In Z one builds abstract values using tuples, sets, relations, functions, sequences, and bags; these all come with pre-defined operations that can be used in assertions. In VDM one has a similar collection of mathematical tools to describe abstract values, and another set of pre-defined operations. In the Larch approach, there are some pre-defined kinds of abstract values (found in Guttag and Horning's LSL Handbook, Appendix A of [**?**]), but these are expected to be extended as needed. (The advantage of being able to extend the mathematical vocabulary is similar to one advantage of object-oriented programming: one can use a vocabulary that is close to the way one thinks about a problem.)

However, there is a problem with using mathematical notations for describing abstract values and their operations. The problem is that such mathematical notations are an extra burden on a programmer who is learning to use a specification language. The solution to this problem is the essential insight that JML takes from the Eiffel lan-

guage [**?**] [**?**] [**?**]. Eiffel is a programming language with built-in specification constructs. It features pre- and postconditions, although it has no direct support for frame axioms. Programmers like Eiffel because they can easily read the assertions, which are written in Eiffel's own expression syntax. However, Eiffel does not provide support for specification-only variables, and it does not provide much explicit support for describing abstract values. Because of this, it is difficult to write specifications that are as mathematically complete in Eiffel as one can write in a language like VDM or Larch/C++.

JML attempts to combine the good features of these approaches. From Eiffel we have taken the idea that assertions can be written in a language that is based on Java expressions. We also adopt the "`old`" notation from Eiffel, which appears in JML as `\old`, instead of the Larch-style annotation of names with state functions. To make it easy to write more complete specifications, however, we use various semantic ideas from model-based specification languages. In particular we use a variant of abstract value specifications, where one describes the abstract value of an object implicitly using several model fields. These specification-only fields allow one to implicitly partition the abstract value of an object into smaller chunks, which helps in stating frame axioms. More importantly, we hide the mathematical notation behind a facade of Java classes. This makes it so the operations on abstract values appear in familiar (although perhaps verbose) Java notation, and also insulates JML from the details of the particular mathematical logic used to do reasoning.

1.8 Acknowledgments

Current version: This current rewrite is largely the work of David R. Cok, Gary Leavens, and Mattias Ulbrich, building on the previous Draft Reference Manual [**?**] and discussions by the JML community at various JML workshops [**?**].

Contributions from David Cok are supported in part by the National Science Foundation: This material is based upon work supported by the National Science Foundation under Grant No. ACI-1314674. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Previous versions:

This is taken from the DRM and needs to be edited/revised/updated.

The work of Leavens and Ruby was supported in part by a grant from Rockwell International Corporation and by NSF grant CCR-9503168. Work on JML by Leavens, and Ruby was also supported in part by NSF grant CCR-9803843. Work on JML by Cheon, Clifton, Leavens, Ruby, and others has been supported in part by NSF grants CCR-0097907, CCR-0113181, CCF-0428078, and CCF-0429567. Support from the NSF continues under a Computing Research Infrastructure (CRI) grant jointly to several institutions: CNS 08-08913 (Leavens at U. of Central Florida, and a subcontact to Rajan

and Basu at Iowa State Unviversity), CNS 07-07874 (Cheon at UTEP), CNS 07-07701 (Clifton at Rose Hulman), CNS 07-07885 (Flanagan at U. Cal. Santa Cruz), CNS 07-08330 (Naumann at Stevens), and CNS 07-09169 (Robby at Kansas State). The work of Poll is partly supported by the Information Society Technologies (IST) Programme of the European Union, as part of the VerifiCard project, IST-2000-26328.

Thanks to Bart Jacobs, Rustan Leino, Peter Müller, Arnd Poetzsch-Heffter, and Joachim van den Berg, for many discussions about the semantics of JML specifications. Thanks for Raymie Stata for spearheading an effort at Compaq SRC to unify JML and ESC/-Java, and to Rustan and Raymie for many interesting ideas and discussions that have profoundly influenced JML.

Thanks to Leo Freitas, Robin Greene, Jesus Ravelo, and Faraz Hussain for comments and questions on earlier versions of this document. Thanks to the many who have worked on the JML checker used to check the specifications in this document. Leavens thanks Iowa State University and its computer science department for helping foster and support the initial work on JML.

See the "Preliminary Design of JML" [**?**] for more acknowledgments relating to the earlier history, design, and implementation of JML.

Chapter 2

Structure of this Manual

Describe overall organization

2.1 Typographical conventions

The remaining chapters of this book follow some common typographical conventions.

This style of text is used for commentary on the JML language itself, such as outstanding issues or now-obsolete practice.

```
Boxed examples
```

comments on how grammars are written; reference to Java grammar

2.2 Grammar

The grammar of JML is intertwined with that of Java. The grammar is given in this Reference Manual as extensions of the Java grammar, using conventional BNF-style productions. The meta-symbols of the grammar are in slightly larger, normal-weight, mono-spaced font. The productions of the grammar use the following syntax:

- non-terminals are written in italics and enclosed in angle brackets: <*expression*>
- terminals, including punctuation as non-terminals, are written in bold typewriter font: **forall** **(** **)** **.**
- parentheses express grouping: (...)

- an infix vertical bar expresses mutually-exclusive alternatives: ... `|` ... `|` ...
- optional elements and repetitions of 0 or more and 1 or more use post-fixed symbols: `?` `*` `+`
- a post-fixed ... indicates a comma-separated list of 0 or more elements: <*expression*> `...`
- a production has the form: <*non-terminal*> `::=` ...

Chapter 3

JML concepts

This chapter describes some general design principles and concepts of the Java Modeling Language. *Say more by way of introduction and motivation*

JML specifications are declarative statements about the behavior of Java entities, namely, packages, classes, methods, and fields. is behavioral good for fields? Typically JML does not make assertions about how a method or class is implemented, only about the net behavior of the implementation. However, to aid in proving assertions about the behavior of methods, JML does include statement and loop specifications.

JML is a versatile specifiation vehicle. It can be used to add lightweight specifications (e.g., specifying ranges for integer values or when a field may hold null) to a program but also to formulate more heavyweight concepts (like abstracting a linked list into a sequence of values).

JML annotations are not bound to a particular tool or approach, but can serve as input to a variety of tools that have different purposes, such as runtime assertion checking, test case generation, extended static checking, full deductive verification, and documentation generation.

To date, most work with JML has been concerned with deductive verification. In that context, specifications and corresponding proof obligations may be considered at different levels of granularities. JML typically is concerned with modular proofs at the level of Java methods. That is, a verification system will establish that each Java method of a program is consistent with its own specifications, presuming the specifications of all methods and classes it uses are correct. If this statement is true for all methods in the program, and all methods terminate, then the system as a whole is consistent with its specifications. *Reference for this claim?*

say more about what it would mean to be modular at a different granularity. Perhaps There should be an entire section on degrees of modularity.

3.1 JML and Java compilation units

A Java program is organized as a set of *compilation units* grouped into packages. The Java language specification does not stipulate a particular means of storing the Java program text that constitutes each compilation unit. However, the vast majority of systems supporting Java programs store each compilation unit as a separate file; the files constituting a package are organized into directory hierarchies corresponding the package and class names.

The simplest way of specifying a Java program with JML is to include the text of the JML specifications directly in the Java source text, as specially formatted comments. By using specially formatted comments to express JML, any existing Java tools will ignore the JML text.

However, in some cases the source Java files are not permitted to be modified or it is preferable not to modify them. Also, the source files may not be available, such as for a Java class library that is shipped only in byte-code form. In these cases, the JML specifications must be expressed separate from the Java source program text in a way that the specifications of packages, classes, methods, and fields can be associated with the correct Java entity.

Thus, specifications written in JML may either (a) be part of the same text that constitutes a Java compilation unit or (b) be in a separate body of text associated with the Java compilation unit. For Java language systems in which Java source material is stored in files, the JML specifications are either in the same file (case (a)) or in a separate file (case (b)). In case (b), the separate file has a `.jml` suffix, the same root name as the corresponding Java source file (typically the name of the public class in the compilation unit), the same package designation, and is stored in the file system's directory hierarchy according to its package and class name, similarly to the Java compilation unit source files. For the rare case in which files are not the basis of Java compilation units, the JML tools must implement a means, not specified here, to recover JML text that is associated with Java source text (case (b) above).

3.2 `.jml` files

MU: I think this section should go into a later chapter, it is to technical here. We mention all JML artifacts before actually having introduced them.

JML files have a `.jml` extension and have a similar appearance to the corresponding `.java` file. The form follows the following rules. Every `.jml` files has a corresponding `.java` or `.class` file; where no `.java` file is available, the rules below refer to the `.java` file that would have been compiled to produce the `.class` file.

The principle present throughout these rules is that declarations in a JML file either (1) correspond to a declaration in the Java file, having the same name, types, non-JML modifiers and annotations, or (2) do not correspond to a Java declaration, and then must

have a different name. Declarations that correspond to a Java declaration must not be in JML comments and must not be marked `ghost` or `model`; JML declarations that do not correspond to Java declarations must be in JML comments and must be marked `ghost` or `model`.

File-level rules

- The `.jml` file has the same package declaration as the `.javafile`.
- The `.jml` file may have a different set of import statements and may, in addition, include model import statements.
- The `.jml` file must include a JML declaration of the public type (i.e., class or interface) declared in the `.java` file. It may but need not have JML declarations of non-public types present in the `.java` class. Any type declared in the `.jml` file that is not present in the `.java` file must be in a JML comment and must have a `model` modifier.

Class declarations

- The JML declaration of a class and the corresponding Java declaration must extend the same superclass, implement the same set of interfaces, and have the same set of non-JML modifiers (`public`, `protected`, `private`, `static`, `final`, *What others*). The JML declaration may add additional JML modifiers or annotations.
- Nested and inner class declarations within an enclosing non-model JML class declaration must follow the same rules as file-level class declarations: they must either correspond in name and properties to a corresponding nested or inner Java class declaration or be a model class.
- JML model classes need not have full implementations, as if they were Java declarations. However, if runtime-assertion checking tools are expected to check or use a model class, it must have a compilable and executable declaration.

Interface declarations

- The JML declaration of a interface and the corresponding Java declaration must extend the extend the same set of interfaces and have the same set of non-JML modifiers (`public`, `protected`, `private`, `static`, *What others*). The JML declaration may add additional JML modifiers or annotations.
- *Comment on static and instance; no initializer for JML field declarations*

Method declarations

- Methods declared in a non-model JML type declaration must either correspond precisely to a method declared in the corresponding Java type declaration or be a model method. *Correspond precisely* means having the same name, exactly the same argument and return types, and the same set of exceptions. MU: look into JLS and check if that is “same signature”

- Methods that correspond to Java methods must not be declared model and must not have a body. They must have the same set of non-JML modifiers and annotations as the Java declaration, but may add additional JML modifiers and annotations.
- A Java method of a class or interface need not have a JML declaration (in which case various default specifications might apply).

Field declarations

- Fields declared in a non-model JML type declaration must either correspond precisely to a field declared in the corresponding Java type declaration or be a model or ghost field. *Correspond precisely* means having the same name and type and non-JML modifiers and annotations. The JML declaration may add additional JML modifiers and annotations.
- A JML field declaration that corresponds to a Java field declaration may not be in a JML comment, may not be `model` or `ghost` and must not have an initializer.
- A JML field declaration that does not correspond to a Java field declaration must be in a JML comment and must be either `ghost` or `model`.
- `ghost` field declarations have the same grammatical form as Java declarations, except that they may use JML types and operators and may refer to names declared in other `ghost` or `model` declarations.
- `model` field declarations have the same grammatical form as Java declarations, except that they may use JML types and operators; they may not have initializers.
- A Java field of a class or interface need not have a JML declaration (in which case various default specifications might apply).

Initializer declarations

- A Java class may contain declarations of static or instance initializers. A JML redeclaration of a Java class may not have any initializers.
- A JML model class may have initializer blocks.

3.2.1 Combining Java and JML files

The specifications for the Java declarations within a Java compilation unit are determined as follows.

- If there is a `.java` file and no corresponding `.jml` file, then the specifications are those present in the `.java` file.
- If there is no `.java` file, but there is a `class` files and a corresponding `.jml` file, then the specifications are those present in the `.jml` file.
- If there is no `.java` file and no `.jml` file, only a `.class` file, then default specifications are used. *Where described?*

If there is a `.java file` and a corresponding to a `.jml` file, then the JML specification present in the `.jml` file supersedes all of the JML specifications in the `.java` file; those in the `.java` file are ignored, even where there is no method declared in the `.jml` file corresponding to a method in the `.java` file. In this case processing proceeds as follows. First all matches among type declarations are established recursively:

- Top-level types in each file are matched by name. The type-checking pass checks that the modifiers, superclass and super interfaces match. JML classes that match are not model and are not in JML comments; JML classes that do not match must be model and must be in JML comments. Not all Java declarations need have a match in JML; those that have no match will have default specifications. The specifications for a Java
- Model types contain their own specifications and are not subject to further matching.
- For each non-model type, matches are established for the nested and inner type declarations in the `.jml` and `.java` declarations by the same process, recursively.

The for each pair of matching JML and Java declarations, matches are established for method and field declarations.

- Field declarations are matched by name. Type-checking assures that declarations with the same name have the same type, modifiers and annotations.
- Method declarations are matched by name and signature. This requires that all the processing of import statements and type declarations is complete so that type names can be properly resolved.

For each pair of matching declarations, the JML specifications present in the `.jml` file give the specifications for the Java entity being declared. If there is a `.jml` file but no match for a particular Java declaration in the corresponding `.java` file, then that declaration uses default specifications, even if the `h.java` file contains specifications. The contents of the `.jml` file supersede all the JML contents of the `.java` file; there is no merging of the files' contents. [1]

Be sure we want this superseding deign rather than merging – could use specs in the Java file if there is no JML declaration, just not merge when both have JML declarations.

3.3 Specification inheritance

Object-oriented programming with inheritance requires that derived classes satisfy the specifications of a parent class, a property known as *behavioral subtyping*[**?**]. Strong behavioral subtyping is a design principle in JML: any visible specification of a parent class is inherited by a derived class. Thus derived types inherit invariants from

[1] Previous definitions of JML did require merging of specifications from multiple files; this requirement added complexity without appreciable benefit. The current design is simpler for tools, with the one drawback that the JML contents of a `.java` file is silently ignored when a `.jml` file is present, even if that `.jml` file does not contain a declaration of a particular entity.

their parent types and methods inherit behaviors from supertype methods they override.

For example, suppose method `m` in derived class `C` overrides method `m` in parent class `P`. In a context where we call method `m` on an object `o` with static type `P`, we will expect the specifications for `P.m` to be obeyed. However, `o` may have dynamic type `C`. Thus `C.m`, the method actually executed by the call `o.m()`, must obey all the specifications of `P.m`. `C.m` may have additional specifications, that is, additional behaviors, constraining its behavior further, but it may not relax any of the specifications given for `P.m`.

Specifications that are not visible in derived classes, such as those marked `private`, are not inherited, because a client cannot be expected to obey specifications that it cannot see. One additional exception to specification inheritance is method behaviors that are marked with the `code` modifier**??**. these behaviors apply only to the method of the class in which the behavior textually appears.

3.4 JML modifiers and Java annotations

The Java Modeling Language was defined prior to the introduction of annotations in Java. Some, but not all, of the features of JML can now be textually represented as Java annotations. Currently JML supports both the old and new syntactic forms.

3.4.1 Modifiers

Modifiers are JML keywords that specify JML characteristics of methods, classes, fields, or variables. Examples are `pure`, `model`, and `ghost`. They are syntactically placed just like Java modifiers, such as `public`.

Each such modifier has an equivalent Java annotation. For example

```
/*@ pure */ public int m(int i)  ...
```

can be written equivalently as

```
@org.jmlspecs.annotation.Pure public int m(int i)  ...
```

The `org.jmlspecs.annotation` prefix can be made implicit in the usual way by including the import statement

```
import org.jmlspecs.annotation.Pure;
@Pure public int m(int i)  ...
```

Note that in the second and third forms, the `pure` designation is now part of the *Java* program and so the import of the `org.jmlspecs.annotation` package must also be in the Java program, and the package must be available to the Java compiler.

All of the modifiers, their corresponding Java annotations, and the locations in which they may be used are described in §F.3.5.

3.4.2 Type modifiers

Some modifiers are actually type modifiers. In particular `non_null` and `nullable` are in this category. Thus the description of the previous subsection (§F.1) apply to these as well.

However, Java 1.8 allows Java annotations to be applied to types wherever type names may appear. For example

```
(@NonNull String)toUpper(s)
```

is allowed in Java 1.8 but is forbidden in Java 1.7.

Need additional discussion of the change in JML for Java 1.8, especially for arrays.

3.5 Model and Ghost

To be written

3.6 Visibility

To be written - note material written in Method Specifications section

3.7 Evaluation and well-formedness of JML expressions

To be written - note material written in Expressions chapter

3.8 Null and non-null references

To be written

Discuss defaults for binary classes; also default specification

Nonnullbydfault is an extension?

3.9 Static and Instance

To be written

3.10 Observable purity

To be written - perhaps this is a separate chapter It might be early days to put this into the standard.

3.11 Location sets and Dynamic Frames

To be written - see section in DRM on Data Groups

3.12 Arithmetic modes

To be written - see later chapter - where shall we put this discussion

3.13 Immutable types and functions

To be written These are the abstract data types I take it.

3.14 Race condition detection

To be written - see later chapter - where shall we put this discussion Should that be part of the standard?

3.15 Redundant specifications

To be written

3.16 Controlling warnings

To be written - nowarn specifications - perhaps in the Syntax chapter; comment on possibility of unsoundness

3.17 org.jmlspecs.lang package

Some JML features are defined in the `org.jmlspecs.lang` package. The `org.jmlspecs.lang` package is included as a model import by default, just as the `java.lang` package is imported by default in a Java file. `org.jmlspecs.lang.*` contains (at least[2]) these elements:

- `JML.informal(<string>)` : This method is a replacement for (and is equivalent to) the informal expression syntax (§**??**) `(* ... *)`. Both expressions return a boolean value, which is always `true`.
- TBD

3.18 Interaction with other tools

3.18.1 Interaction with Type Annotations in Java 1.8

To be written

3.18.2 Interaction with the Checker framework

To be written

3.18.3 Interaction with FindBugs

To be written

3.19 Core JML

This standardization document describes all of JML. However, some portions of the language are considered *Core*, whereas others might be conveniences, are rarely used or applicable only in less common situations. This leaves it possible that some tools might only implement the Core.

The following table identifies the Core functionality and indicates which tools implement which items.

Also some syntactical redundancy features are not part of the core.

The entries in the table have these meanings:

[2]Tools implementing JML may add additional methods.

- **Core**– an JML construct in the Core
- 1 – part of level 1 above Core
- **Ext**- an extension to JML (not defined as standard)
- **Dep**- deprecated features of JML
- **–**– not supported by tool
- **– –**– not supported by tool
- **+**– supported by tool
- **ESC**– supported by tool for static checking only (not runtime)
- **RAC**– supported by tool for runtime checking only (not static)
- **++**- supported by tool

This table is being edited and is not correct

keyword	Core	KeY	OpenJML	Comments
`<==`	1	+	+	
`<==>`	**Core**	++	+	
`==>`	**Core**	++	+	
`//@`	**Core**	++	+	
`/*@ @*/`	**Core**	++	+	
`(* *)`	1	– –	+	MU: If we find a good semantics
`accessible`	+	+	–	
`also`	+	++	+	
`assert`	+	++	+	
`assignable`	+	++	+	KeY: also for loops!
`assume`	+	+	+	
`axiom`	+	+	+	
`behavior`	+	++	+	MU: BrE is syntactic sugar
`\\bigint`	+	++	+	
`code`	+	–	+	
`constraint`	+	+	+	
`\count`	+	–	+	
`decreases`	+	++	+	
`diverges`	+	++	+	
`\dl_`	–	–		MU: or some other means of tool-spec exts.
`\elemtype`	+	+		
`ensures`	+	++		
`\everything`	+	+		
`exceptional_behavior`	+	++		

keyword	Core	KeY	OpenJML	Comments
`\exists`	+	++		
`\forall`	+	++		
`\fresh`	+	++		
`ghost`	+	++		
`helper`	+	++		
`\\index`	+	++		the new addition discussed in Bad Her.
`initially`	+	?		
`instance`	+	−−		
`\invariant_for`	+	++		
`\locset`	+	++		builtin datatype
`loop_invariant`	+	++		`maintains` might be more systematic
`\max`	+	++		semantics interesting
`measured_by`	++		generalised version with list of items	
`\min`	+	++		
`model`	+	++		fields, methods
`model`	−	−		classes
`model`	−	−−		import statements
`non_null`	+	++		
`\nonnullelements`	+	+		
`normal_behavior`	+	++		
`no_state`	−	+		
`\nothing`	+	++		
`nullable`	+	++		
`nullable_by_default`	+	++		
`\num_of`	+	+		
`\old`	+	++		w/o label
`\old`	−	+		w/ label
`private`	+	++		for invariants and contracts
`\product`	+	++		
`protected`	+	−		for invariants and contracts. Meaning must be clarified.
`pure`	+	++		@Pure is syntactic sugar
`\real`	+	++		
`represents`	+	++		
`requires`	+	++		
`\result`	+	++		
`signals`	+	++		
`signals_only`	+	++		
`spec_bigint_math`	+	+		
`spec_java_math`	+	+		

keyword	Core	KeY	OpenJML	Comments
`spec_protected`	+	++		
`spec_public`	+	++		
`spec_safe_math`	+	–		
`\static_invariant_for`	–	++		
`\strictly_nothing`	**Ext**	–	+	
`strictly_pure`	**Ext**	–	+	
`\sum`	1	++		
`two_state`	**Ext**	+	–	
`\type`	1	–	+	
`\TYPE`	1	–	+	
`\typeof`	**Core**	–	+	
`\values`	**Core**	+	+	
block contracts	1	++	+	
annotations instead of modifiers	1	–	+	I would not add them to the core, but explain them only in an appendix on syntactical variations. It should be added to the core when annotations are an alternative for all specifications
`//` comments in specs	1	++	+	
`//` JML in Javadoc	**Dep**	–	–	
`{| ... |}`	–		not widely used, is it?	

New primitive datatype `\locset` with the following operators: (Reification of data-groups / regions)

- `\nothing` only existing locations
- `\everything` all locations
- `\empty` no location at all: The empty set.
- `\union`(...) arbitrary arity
- `\intersect`(...) arbitrary arity
- `\minus`($\cdot,\cdot$)
- `\subset`($\cdot,\cdot$)
- `\disjoint`(...) pairwise disjointness

- `(\collect ...; ...; ...)` a variable binder in the sense of

$$\bigcup_{x|\varphi} locs(x) = \texttt{(\collect}\ T\ x\texttt{;}\ \varphi\texttt{;}\ locs(x)\texttt{)},$$

  e.g., `(\collect int i; 0<=i && 2*i<a.length; a[2*i])` is the set of all locations in $a[*]$ with even index.

  Often needed for things like `(\collect Person p; set.contains(p); p.footprint)`.

- `\new_elements_fresh`$(\cdot)$ with the meaning

$$\texttt{\new\_elements\_fresh}(ls) := \forall l \in ls.l \in \texttt{\old}(ls) \vee \texttt{\fresh}(object(l))$$

  . This is used to confine the extension of a location set in a postcondition to objects which have been recently created. This is important to guarantee framing in dynamic frame specifications. This is sometimes called the *swinging pivot* property. (Reasoning is usually: If ls_1 and ls_2 are disjoint before a method and both ls_1 is not touched and ls_2 grows only into fresh objects, then ls_1 and ls_2 are still disjoint after the method.)

Chapter 4

JML Syntax

These first two sections need to be merged

4.1 Textual form of JML specifications

JML text is expressed in specially-formatted Java comments. Java comments either

- (a) begin with the characters `//` and extend through the end of the line or
- (b) begin with the characters `/*` and extend through the next occurrence of the characters `*/`, possibly spanning multiple lines.

Unconditional JML text either

- (a) begins with the text `//@` and extends through the end of the line or
- (b) begins with the text `/*@` and extends through the next occurrence of `*/`.

Any `@` symbols within JML text and not inside string or character constants are considered white space. Typical uses of such white space `@` symbols are at the beginning of JML text, the end of text or the beginnings of lines within text, as shown in the example below. Only the use at the beginning of lines or just before the closing `*/` is common (or suggested).

```
/*@@@@@@@@@@@@@@@@@@@@@@@@@@@@
  @   requires x > 0;
  @   ensures \result < 0;
  @@@@@@@@@@@@@@@@@@@@@@@@@@@@@*/
```

Conditional JML text includes some non-whitespace characters between the `//` or `/*` and the `@` at the beginning of the comment. That text has the syntax `([+|-]<java-identifier>)+`, that is, one or more Java-identifiers, each preceded by either a `+` or a `-` . No white space is permitted between the initial `//` or `/*` and the `@` that follows the conditional keys. A

Java-identifier is a sequence of alphanumeric characters, underscores and dollar signs, that does not begin with a digit.

The conditional JML text feature presumes that the JML tool processing the JML text has a means to define various identifiers for the purpose of selecting or deselecting JML text. An identifier occurring before the `@` in a given JML comment is *positive* if it is directly preceded by a + sign and negative if directly preceded by a - sign. The JML text in a conditional comment is included or ignored according to this rule.

- A conditional JML comment is included if (and only if) none of its negative identifiers (if any) are defined and there is at least one positive identifier that is defined.

Here are some examples, presuming the JML tool has defined the identifiers `OPTA` and `OPTB`, but not the identifier `OPTC`.

```
/*@ ... */  -- included unconditionally
/*+OPTA@ ... */ -- included because OPTA is defined and is positive
/*+OPTC@ ... */ -- ignored because OPTC is not defined
/*-OPTA@ ... */ -- ignored because OPTA is defined and but is negative
/*-OPTC@ ... */ -- ignored because OPTC is not defined
/*+OPTA+OPTC@ ... */ -- ignored because OPTA is defined and positive, even though OPTC is not defined
/*+OPTA-OPTC@ ... */ -- ignored because OPTA is defined and positive, because though OPTC is negative it is not de
/*+OPTA-OPTB@ ... */ -- ignored because OPTB is defined and is negative, even though OPTA is defined and positive
```

JML reserves the following identifiers:

- `DEBUG` : not defined by default; when defined, the JML `debug` statement is enabled
- `ESC` : tools should define this identifier when doing static checking
- `RAC` : tools should define this identifier when doing runtime checking

---

Issues with the JML textual format There are two issues that can arise with the syntactical design of JML. First, other tools may also use the `@` symbol to designate comments that are special to that tool. If JML tools are trying to process files with such comments, the tolls will interpret the comments as JML comments, likely causing a myriad of parsing errors.
Second, Java uses the `@` sign to designate Java annotations. That in itself is not an ambiguity, but sometimes users will comment out such annotations with a simple preceding `//`, as in

```
//@MyAnnotation
```

This construction now looks like JML. The solution is to be sure there is whitespace between the `//` and the `@`, but it may not always be possible for the user to perfom such edits.

4.2 JML Syntax

4.2.1 Syntax of JML specifications

JML specifications may be written as Java annotations. Currently these are only implemented for modifiers (cf. section TBD). In Java 8, the use of Java annotations for JML features will be expanded.

JML specifications may also be written in specially formatted Java comments: a JML specification includes everything between either (a) an opening `/*@` and closing `*/` or (b) an opening `//@` and the next line ending character (`\n` or `\r`) that is not within a string or character literal.

Such comments that occur within the body of a class or interface definition are considered to be a specification of the class, a field, or a method, depending on the kind of specification clause it is. JML specifications may also occur in the body of a method.

Obsolete syntax. In previous versions of JML, JML specifications could be placed within javadoc comments. Such specifications are no longer standard JML.

4.2.2 Conditional JML specifications

JML has a mechanism for conditional specifications, based on a system of keys. A key is an identifier (consisting of ASCII alphanumeric and underscore characters, and beginning with a non-digit). A conditional JML comment is guarded by one or more positive or negative keys (or both). The keys are placed just before the `@` character that is part of the opening sequence of the JML comment (the `//@` or the `/*@`). Each key is preceded by a '+' or a '-' sign, to indicate whether it is a positive or negative key, respectively. *No white-space is allowed*. If there is white-space anywhere between the initial `//` or `/*` and the first `@` character, the comment will appear to be a normal Java comment and will be silently ignored.

The keys are interpreted as follows. Each tool that processes the Java+JML input will have a means (e.g., by command-line options) to specify the set of keys that are enabled.

- If the JML annotation has no keys, the annotation is always processed.
- If there are only positive keys, the annotation is processed only if at least one of the keys is enabled.
- If there are only negative keys, the annotation is processed unless one of the keys is enabled.

- If there are both positive and negative keys, the annotation is processed only if (a) at least one of the positive keys is enabled AND (b) none of the negative keys are enabled.

JML previously defined one conditional annotation: those that began with `/*+@` or `//+@`. ESC/Java2 also defined `/*-@` and `//-@`. Both of these are now deprecated.

JML defines the following keys, which conforming tools must support. Tools may also define keys of their own and should silently ignore keys they do not recognize. It is expected that each tool will define at least one key that can enable any extensions that tool implements or disable JML features not supported by the tool.

- **ESC** : the `ESC` key is enabled when a tools is performing ESC static checking;
- **RAC** : the `RAC` key is enabled when a tool is compiling for Runtime-Assertion-Checking.
- **DEBUG** : The `DEBUG` key is not implicitly enabled. However it is defined as the key that enables the **debug** JML statement. That is the **debug** statement is ignored by default and is used by tools if the user enables the `DEBUG` key.
- **OPENJML** : The OPENJML key is enabled whenever the OpenJML tool is processing annotations (and presumably is not enabled by other tools).
- **KEY**: The `KEY` key is reserved for use by the KeY tool [**?**], and should be ignored by other tools.

Thus, for example, one can turn off a non-executable assert statement for RAC-processing but retain it for ESC and for type-checking by writing //-RAC@ assert ...

Chapter 5

JML Types

All of the Java type names are legal and useful in JML: `int short long byte char boolean double real` and class and interface types. In addition, JML defines some additional types, described in subsections below. There are two needs that JML addresses:

- Specifications are sometimes best written using infinite-precision mathematical types, rather than the fixed bit-width types of Java. JML's arithmetic modes (§**??**) allow choosing among various numerical precisions.
- Java's handling of class types only expresses erased types; JML adds a type and operations for expressing and reasoning about generic types.

Location set type? Object set type?

5.1 `\bigint`

The `\bigint` type is the set of mathematical integers (i.e., $\mathbb{Z}$). Just as Java primitive integral types are implicitly converted to `int` or `long`, all Java primitive integral types implicitly convert to `\bigint` where needed. When `\bigint` values need to be autoboxed into an Object, they are boxed as `java.math.BigInteger` values; similarly when JML specifications are compiled for runtime checking, `\bigint` values are represented as `java.math.BigInteger` values. Within JML specifications, however, the `\bigint` type is treated as a primitive type.

For example, `==` with two `\bigint` eperands expresses equality of the represented integers, not (Java) identity of `BigInteger` objects.

The familiar operators are defined on values of the `\bigint` type: unary and binary `+` and `-`, `*`, `/`, `%`. Also, these types can be used in quantified expressions and variables of these types can be declared as ghost or model variables.

5.2 \real

The \real type is the set of mathematical real numbers (i.e., $\mathbb{R}$). Just as the Java primitive type float is implicitly converted to double, both real and double values implicitly convert to \real where needed. When \real values need to be auto-boxed into an Object, they are boxed as ???TODO values; similarly when JML specifications are compiled for runtime checking, \real values are represented as ???TODO values. Within JML specifications, however, the \real type is treated as a primitive type. Integral values, including \bigint, are implicitly converted to \real where necessary.

The familiar operators are defined on values of the \real type: unary and binary + and -, *, /, %. Also, these types can be used in quantified expressions and variables of these types can be declared as ghost or model variables.

5.3 \TYPE

TODO

OLD STUFF:

The set of \TYPE values includes non-generic types such has \type(org.lang.Object), fully parameterized generic types, such as \type(org.utils.List<Integer>), and primitive types, such as \type(int). The subtype operator (<:) is defined on values of type \TYPE.

TBD - what about other constructors or acccessors of TYPE values

Chapter 6

JML Specifications for Packages and Compilation Units

There are no JML specifications at the package level. If there were, they would likely be written in `package-info.java` file. The only JML specifications that are defined at the file level, applying to all classes defined in the file, are model import statements.

6.1 Model import statements

Add in top-level model classes

Java's import statements allow class and (with static import statements) field names to be used within a file without having to fully qualify them. The same import statements apply to names in JML annotations. In addition, JML allows *model import* statements. The effect of a JML model import statement is the same as a Java import statement, except that the names imported by the JML statement are only visible within JML annotations. If the model import statement is within a `.jml` file, the imported names are visible only within annotations in the `.jml` file, and not outside JML annotations and not in a corresponding `.java` file. These are import statements that only affect name resolution within JML annotations and are ignored by Java. They have the form

`//@ model` <*Java import statement*>

Note that the Java import statement ends with a semicolon.

Note that both

`model` <*Java import statement*>`;`

and

```
/*@ model */<Java import statement>;
```

are invalid. The first is not within a JML comment and is illegal Java code. The second is a normal Java import with a comment in front of it that would have no additional effect in JML, even if JML recognized it (tools should warn about this erroneous use).

6.2 Default imports

The Java language stipulates that `java.lang.*` is automatically imported into every Java compilation unit. Similarly in JML there is an automatic model import of `org.jmlspecs.lang.*`. However, there are not yet any standard-defined contents of the `org.jmlspecs.lang` package. Is this correct?

6.3 Issues with model import statements

As of this writing, no tools distinguish between Java import statements and JML import statements. Such implementations may resolve names in Java code differently than the Java compiler does. Consider two packages `pa` and `pb` each declaring a class `N`.

1)

```
import pa.N;
//@ model import pb.N;
```

Correct behavior: In Java code `N` is `pa.N`; in JML code, `N` is ambiguous.
Non-conforming behavior: JML tools consider `N` in Java code to be ambiguous.

2)

```
import pa.N;
//@ model import pb.*;
```

Correct behavior: In Java code `N` is `pa.N`; in JML code, `N` is `pa.N`.
Non-conforming behavior: non-conforming JML tools will act correctly in this case.

3)

```
import pa.*;
//@ model import pb.N;
```

Correct behavior: In Java code `N` is `pa.N`; in JML code, `N` is `pb.N`.
Non-conforming behavior: JML tools consider `N` in Java code to be `pb.N`.

4)

```
import pa.*;
//@ model import pb.*;
```

Correct behavior: In Java code `N` is `pa.N`; in JML code, `N` is ambiguous.
Non-conforming behavior: JML tools consider `N` in Java code to be ambiguous.

Chapter 7

Specifications for Java types in JML

By *types* in this reference manual we mean classes, enums, and interfaces, whether global, secondary, local, or anonymous. Some aspects of JML, such as the allowed modifiers, will depend on the kind of type being specified.

Need to work out specs for enum types

Specifications at the type level serve three different primary purposes: specifications that are applied to all methods in the type, specifications that state properties of the data structures in the type, and declarations that help with information hiding.

7.1 Modifiers for type declarations

Modifiers are placed just before the construct they modify. Example Java modifiers are `public` and `static`. JML modifiers may be in their own annotation comments or grouped with other modifiers, as shown in the following example code.

```
//@ pure
public class C {...}

public /*@ pure nullable_by_default */ class D {...}
```

7.1.1 non_null_by_default, nullable_by_default, @NonNullByDefault, @NullableByDefault

The `non_null_by_default` and `nullable_by_default` modifiers or, equivalently, the `@NonNullByDefault` and `@NullableByDefault` Java annotations, specify the default nullity declaration within the class. The default applies to all field and local variable declarations, to formal parameters and method return values, and recursively to any nested or inner classes that do not have default nullity declarations of their own.

These default nullity modifiers are not inherited by derived classes.

A class cannot be modified by both modifiers at once. If a class has no nullity modifier, it uses the nullity modifier of the enclosing class; the default for a top-level class is `non_null_by_default`. This top-level default may be altered by tools.

7.1.2 pure and @Pure

Specifying that a class is *pure* means that each method and nested class within the class is specified as pure. The `pure` modifier on a class is not inherited by derived classes, though `pure` modifiers on methods are.

There is no modifier to disable an enclosing `pure` specification.

7.1.3 @Options

The `@Options` modifier takes a String argument, which is a string of command-line options and corresponding arguments. These command-line options are applied to the processing (e.g., ESC or RAC) of each method within the class. The options may be augmented or disabled by corresponding `@Options` modifiers on nested methods or classes. In effect, the options that apply to a given class are the concatenation of the options given for each enclosing class, from the outermost in.

An Options modifier is not inherited by derived classes.

Or does Option take an array of String

7.2 invariant clause

TODO

7.3 constraint clause

TODO

7.4 initially clause

Grammar:

An `initially` clause for a type is equivalent to an additional postcondition for each constructor of the type, as if an additional `ensures` clause (with the predicate stated by the `initially` clause) is added to every behavior of each constructor in the type.

Say more about initially clauses in interfaces and enums

7.5 ghost fields

TODO

7.6 model fields

TODO

7.7 represents clause

TODO

7.8 model methods and model classes

TODO

7.9 initializer and static_initializer

TODO

7.10 axiom

TODO

7.11 readable if clause and writable if clause

TODO

7.12 monitors_for clause

TODO

Chapter 8

JML Method specifications

Method specifications describe the behavior of the method. JML is a modular specification methodology, with the Java method being the fundamental unit of modularity. The specifications may under-specify a method. For example, the specifications may simply say that the method always returns normally (that is, without throwing an exception), but give no constraints on the value returned by the method. The degree of precision needed will depend on the context.

8.1 Structure of JML method specifications

```
<method-spec> ::= ( also )?  <behavior-seq>
                    ( also implies_that <behavior-seq> )?
                    ( also for_example <behavior-seq> )?

<behavior-seq> ::= <behavior> ( also <behavior> )*

<behavior> ::=
          ( ( <java-visibility> ( code )?  <behavior-id> )?  <clause-seq>)
        | <java-visibility> ( code )?  <model-program>

<java-visibility> ::= ( public | protected | private )?

<behavior-id> ::= behavior | normal_behavior | exceptional_behavior
                    | behaviour | normal_behaviour | exceptional_behaviour

<clause-seq> ::= ( <clause> | <nested-clause> )+

<clause> ::=
          <requires-clause>
       | <assignable-clause>
       | <ensures-clause>
       | <signals-clause>
```

```
    | <signals-only-clause>
    | <diverges-clause>
    | <measured-by-clause>
    | <duration-clause>
    | <when-clause>
    | <old-clause>
    | <forall-clause>
    | <working-space-clause>
    | <accessible-clause>
    | <callable-clause>
    | <captures-clause>
    | <method-program-block>
```

```
<nested-clause> ::= {| <clause-seq> ( also <clause-seq> )* |}
```

Meta-parser rules:

- Each of the behavior keywords spelled **behaviour** is equivalent to the corresponding keyword spelled **behavior**. The latter is more common.
- A behavior beginning with **normal_behavior** may not contain a <*signals-clause*> or a <*signals-only-clause*>. It implicitly contains the clause **signals (Exception e) false;**.
- A behavior beginning with **exceptional_behavior** may not contain a <*ensures-clause*>. It implicitly contains the clause **ensures false;**.
- Clause ordering: in any consecutive sequence of <*clause*>, any <*old-clause*> and <*forall-clause*> must appear before any <*requires-clause*>, which must appear before any other clauses. It is a JML extension to allow clauses in any order. However, even in that case:
- The order of <*requires-clause*> and <*old-clause*> clauses is significant in the same way that the order of terms in a short-circuit boolean expression is significant: earlier <*requires-clause*> expressions may state conditions that enable later ones to be well-defined.
- Any declarations in <*old-clause*> and <*forall-clause*> clauses must precede any uses of the declared variables.
- The order of <*ensures-clause*>s is also significant in that earlier expressions can assert conditions that are required for later expressions to be well-defined.

  Is this a requirement or a style recommendation?
- *May any clauses appear after a nested-clause?*

Note that the vertical bars in the production for *nested-clause* are literals, not meta-symbols.

OK to relax the rules on which clauses can be present where in parsing, and then enforce or advise in later checking?

FIXME - the method-spec production is not correct - the first also might be omitted whichever one it is

8.1.1 Behaviors

The basic structure of JML method specifications is as a set of *behaviors*. Each behavior contains a set of *clauses*. The various kinds of clauses are described in the subsequent sections of this chapter. Each kind of clause has a default, which applies if the clause is textually absent from the behavior. A particular kind of clause is the `requires` clause, which gives the *precondition* for the behavior; the other clauses state the properties that are true about the method's execution.

For each behavior, if the method is called in a context in which the behavior's precondition is true, then the method must adhere to the constraints specified by the remaining clauses of the behavior. Not all of the preconditions need be true, but at least one of them must be, otherwise the method is being called in a context in which its behavior is undefined. For example, a method's specification may have two behaviors, one with a precondition that the method's argument is not null, and the other behavior with a precondition that the method's argument is null. In this case, in any context, one or the other behavior will be active. If however, the second behavior were not specified, then it would be a violation to call the method in any context other than those in which the first precondition, that the argument is not null, is true. More than one behavior may be active (have its precondition true); every active behavior must be obeyed by the method. Where preconditions are not mutually exclusive, care must be taken that the behaviors themselves are not contradictory, or it will not be possible for any implementation to satisfy the combination of behaviors.

8.1.2 Nested specification clauses

Nested specification clauses are syntactic shorthand for an expanded equivalent in which clauses are replicated. The nesting syntax simply allows expressing common subsequences of clauses to be expressed without repetition, where that helps understandability.

In particular, referring to the grammar above, a <*behavior*> whose <*clause-seq*> contains a <*nested-clause*> is equivalent to a sequence of <*behavior*>s as follows:
if <*nested-clause*>$_A$ is a combination of n <*clause-seq*> as in
{| <*clause-seq*>$_{S1}$ (**also** <*clause-seq*>$_{Si}$)* |}
then
(<*java-visibility*>$_V$(**code**)?$_W$<*behavior-id*>$_X$)?<*clause*>*$_D$<*nested-clause*>$_A$<*clause-seq*>$_E$
is equivalent to a sequence of n <*behavior*> constructions
(<*java-visibility*>$_V$(**code**)?$_W$<*behavior-id*>$_X$)?<*clause*>*$_D$<*clause-seq*>$_{S1}$<*clause-seq*>$_E$
also
...

also
(<*java-visibility*>$_V$(**code**)?$_W$<*behavior-id*>$_X$)?<*clause*>∗$_D$<*clause-seq*>$_{Sn}$<*clause-seq*>$_E$

Is there a better way to describe this desugaring? and a better way to format it?

8.1.3 Specification inheritance and the `code` modifier

The behaviors that apply to a method are those that are textually associated with the method (that is, they precede the method definition in the .java or .jml file) and those that apply to methods overridden by the given method. In other words, method specifications are inherited (with exceptions given below), as was described in §3.3.

Specification inheritance has important consequences. A key one relates to preconditions. The composite precondition for a method is the *disjunction* of the preconditions for each behavior, including the behaviors of overridden methods. Thus, just looking at the behavior within a method, one might not immediately realize that other behaviors are permitted and that the precondition is more accepting.

There are a few cases in which behaviors are not inherited:

- Since static methods are not overridden, their behaviors are also not inherited.
- Since private methods are not overridden, their behaviors are also not inherited.
- `private` behaviors are not inherited.
- Behaviors marked with the `code` modifier are not inherited.

The `code` modifier is unique in that it applies to method behaviors and nowhere else in JML. It is specifically used to indicate that the behavior is not inherited. The `code` modifier is allowed but not necessary if the behavior would not be inherited anyway. The `code` modifier is not allowed if the method does not have a body; so it is not used on an abstract method declaration, unless that method is marked `default` (in Java) and has a body.

Java allows a class to extend multiple interfaces. More than one interface might declare behaviors for the same method. An implementation of that method inherits the behaviors from all of its interfaces (recursively) .

8.1.4 Visibility

The following discussion has some errors and needs fixing; also need to talk about spec_public, spec_protected

Each method specification behavior has a *java-visibility* (cf. the discussion in §**??**). Any of the kinds of behavior keywords (`behavior`, `normal_behavior`, `exceptional_behavior`) may be prefixed by a Java visibility keyword (`public`, `protected`, `private`); the absence of a visibility keyword indicate package-level visibility. A lightweight behavior (one without a behavior keyword) has the visibility of its associated method.

Table 8.1: Visibility rules for method specification behaviors

Behaviors with this visibility	may contain names that are visible in the class because of this visibility
public	public
protected	public, protected-by-inheritance
package	public, protected-by-package, package
private	any

The visibility of a behavior determines the names that may be referenced in the behavior. The general principle is that a client that has permission to see the behavior must have permission to see the entities in the behavior. Thus

```
any name (of a type, method or field) in a method specification
visible to a client must also be visible to the client.
```

For example, a public behavior may contain only public names. A private behavior may contain any name visible to a client that can see the private names; this would include other private entities in the same or enclosing classes, any public name, any protected name from super classes, and any package or protected name from other classes in the same package. The visibility for protected and package behaviors is more complex. A protected behavior is visible to any client in the same or subclasses; since the subclasses may be in a different package, the protected behavior may contain other names with protected visibility only if they are visible in the behavior by virtue of inheritance, and not if the are visible only because of being in the same package. To be explicit, suppose we have class A, unrelated class B in the same package, class C a superclass of A in a different package, and class D derived from A but in a different package, with identifiers A.a, B.b, and C.c each with protected visibility. Only A.a and C.c are visible in class D; thus a protected behavior in class A, which is visible to D, may contain A.a and C.c but not B.b. Similarly a behavior with package visibility may only contain names that are visible by virtue of being in the same package (and public names); names with protected visibility that are visible in a class by virtue of inheritance are not necessarily visible to clients who can see the package-visible behavior.

The root of the complexity is that protected visibility is not transitive, whereas the other kinds of Java visibility are. Conceptually, protected visibility must be separated into two kinds of visibility: protected-by-inheritance and protected-by-package. Each of these is separately transitive. Then the visibility rules can be summarized in Table 8.1.

8.1.5 Grammar of method specifications

Fillin – remember lightwieght, behavior, normal_behavior, exceptional_behavior, examples, implies_that, visibility, model program behaviors, `also`, nested behaviors

Do we relax the ordering and the constraints on nesting that are in the current RefMan

Comment on comparison with ACSL

8.2 Method specifications as Annotations

8.3 Modifiers for methods

TODO

8.4 Common JML method specification clauses

TODO

8.4.1 `requires` clause

TODO

8.4.2 `ensures` clause

TODO

8.4.3 `assignable` clause

TODO

8.4.4 `signals` clause

TODO

8.4.5 `signals_only` clause

TODO

8.5 Advanced JML method specification clauses

TODO

8.5.1 `accessible` clause

TODO

8.5.2 `diverges` clause

TODO

8.5.3 `measured_by` clause

TODO

8.5.4 `when` clause

TODO

8.5.5 `old` clause

TODO

8.5.6 `forall` clause

TODO

8.5.7 `duration` clause

TODO

8.5.8 `working_space` clause

TODO

8.5.9 `callable` clause

TODO

8.5.10 `captures` clause

TODO

8.6 Model Programs (`model_program` clause)

8.6.1 Structure and purpose of model programs

8.6.2 `extract` clause

TODO

8.6.3 `choose` clause

TODO

8.6.4 `choose_if` clause

TODO

8.6.5 `or` clause

TODO

8.6.6 `returns` clause

TODO

8.6.7 `continues` clause

TODO

8.6.8 `breaks` clause

TODO

8.7 Modifiers for method specifications

8.7.1 pure and @Pure

TBD

8.7.2 non_null, nullable, @NonNull, and @Nullable

TBD

8.7.3 model and @Model

TBD

8.7.4 spec_public, spec_protected, @SpecPublic, and @SpecProtected

These modifiers apply only to methods declared in Java code, and not to methods declared in JML, such as model methods.

TBD

8.7.5 helper and @Helper

TBD

8.7.6 function and @Function

TBD

8.7.7 query, secret, @Query, and @Secret

TBD

8.7.8 code_java_math, code_bigint_math, code_safe_math

TBD - add rest and annotations

8.7.9 skip_esc, skip_rac, @SkipEsc, and SkipRac

These modifiers apply only to methods with bodies.

When these modifiers are applied to a method or constructor, static checking (respectively, runtime checking) is not performed on that method. In the case of RAC, the method will be compiled normally, without inserted checks. These modifiers are a convenient way to exclude a method from being processed without needing to remember to use the correct command-line arguments.

8.7.10 options and @Options

This modifier takes one or more string literals as arguments. The literals a *Get format correct - one option per argument?*

8.7.11 extract and @Extract

This modifier applies only to methods with bodies.

TBD

8.8 TODO Somewhere

constructor field method nowarn

<:: token

lots more backslash tokens

Chapter 9

Field Specifications

Fields may have various modifiers, each of which states a restriction on how the field may be used. Fields may be part of *data groups*, which allow specifying frame conditions on fields that may not be visible because of the Java visibility rules. Also, a specification may introduce *ghost* or *model* fields that are used in the specification but are not present in the Java program.

9.1 Field and Variable Modifiers

The modifiers permitted on a field, variable, or formal parameter declaration are shown in Table 9.1.

9.1.1 non_null and nullable (@NonNull, @Nullable)

The non_null and nullable modifiers, and equivalent @NonNull and @Nullable annotations, specify whether or not a field, variable, or parameter may hold a null value. The modifiers are valid only when the type of the modified construct is either a reference or array type, not a primitive type.

Discuss @NonNull TestJava a, b;

9.1.2 spec_public and spec_protected (@SpecPublic, @SpecProtected)

Need to discuss relationship with JSR308

These modifiers are used to change the visibility of a Java field when viewed from a JML construct. A construct labeled `spec_public` has `public` visibility in a JML specification, even if the Java visibility is less than public; similarly, a construct labeled `spec_protected` has `protected` visibility in a JML specification, even if the Java

Table 9.1: Modifiers allowed on field, variable and parameter declarations

Modifier	Where	Purpose
non_null	field, var, param	the variable may not be null (§9.1.1)
nullable	field, var, param	the variable may be null
spec_public	field	visibility is public in specs
spec_protected	field	visibility is protected in specs
model	field	representation field
ghost	field, var	specification only field
uninitialized	var	TBD
instance	field	not static
monitored	field	guarded by a lock
secret	field, var, param	hidden field
peer	field, param	TBD
rep	field, param	TBD
readonly	field, param	TBD

visibility is less than protected. Section **??** contains a detailed discussion of the effect of information hiding using Java visibility on JML specifications.

Listing 9.1: Use of spec_public

```
private /*@ spec_public */ int value;

//@ ensures value == i;
public setValue(int i) {
  value = i;
}
```

For example, Listing 9.1 shows a simple setter method that assigns its argument to a private field named `value`. The visibility rules require that the specifications of a public method (`setValue`) may reference only public entities. In particular, it may not mention `value`, since `value` is private. The solution is to declare, in JML, that `value` is `spec_public`, as show in the Listing.

9.1.3 `ghost` and `@Ghost`

9.1.4 `model` and `@Model`

9.1.5 `uninitialized` and `@Uninitialized`

9.1.6 `instance` and `@Instance`

9.1.7 `monitored` and `@Monitored`

9.1.8 `query`, `secret` and `@Query`, `@Secret`

9.1.9 `peer`, `rep`, `readonly` (`@Peer`, `@Rep`, `@Readonly`)

Check readonly vs. read_only, Readonly vs. ReadOnly

9.2 Datagroups: `in` and `maps` clauses

TODO

9.3 Ghost fields

TODO

9.4 Model fields

TODO

Chapter 10

JML Statement Specifications

JML Statement specifications are JML constructs that appear as statements within the body of a Java method or initializer. Some are standalone statements, while others are specifications for loops or blocks that follow.

Grammar:

```
<jml-statement> ::=
                          <jml-assert-statement>
                        | <jml-assume-statement>
                        | <jml-local-declaration>
                        | <jml-unreachable-statement>
                        | <jml-reachable-statement>
                        | <jml-set-statement>
                        | <jml-debug-statement>
                        | <jml-loop-specification>
                        | <jml-hence-by-statement>
                        | <jml-refining-specification>
```

10.1 `assert` statement and Java assert statement

Grammar:

`<jml-assert-statement>` ::= **assert** `<jml-expression>` **;**

Type checking requirements:

- the <*jml-expression*> must be boolean

The `assert` statement that the given expression must be `true` at that point in the program. A static checking tool is expected to require a proof that the asserted expression is true and to issue a warning if the expression is not provable. A runtime assertion checking tool is expected to check whether the asserted expression is true and to issue a warning message if it is not true in the given execution of the program.

Note that a JML assert statement has a different effect than a Java assert statement. When assertion checking is enabled, a Java assert statement will result in a AssertionError at runtime if the corresponding assertion evaluates to false; if assertion checking is disabled (the default), a Java assert statement is ignored. Runtime assertion checking tools may implement JML assert statements as Java assert statements.

10.2 `assume` statement

Grammar:

<jml-assume-statement> ::= **assume** *<jml-expression>* **;**

Type checking requirements:

- the <*jml-expression*> must be boolean

The `assume` statement adds an assumption that the given expression is `true` at that point in the program.

Static analysis tools may assume the given expression to be true. Runtime assertion checking tools may choose to check or not to check the assume statements.

An `assume` statement might be used to state an axiom or fact that is not easily proved. However, `assume` statements should be used with caution. Because they are assumed but not proven, if they are not actually true an unsoundness will be introduced into the program. For example, the statement `assume false;` will render the following code silently infeasible. Even this may be useful, since, during debugging, it may be helpful to shut off consideration of certain branches of the program.

10.3 ghost and model declarations

10.4 `unreachable` statement

Grammar:

<jml-unreachable-statement> ::= **unreachable** *<jml-expression>?* **;**

Type checking requirements:

- the optional <*jml-expression*> must be boolean; if not present its default value is **true**.

The `unreachable` statement asserts that no feasible execution path will ever reach this statement when the given expression is true.

The terminating semicolon is required and is easily forgotten, since statement is almost always used without the optional expression.

The unreachable statement with an expression is an extension to standard JML.

It has been common practice to insert `assert false;` statements to check whether a given program point is infeasible. The `unreachable` statement accomplished the same purpose with clearer syntax.

10.5 `reachable` statement

Grammar:

<jml-reachable-statement> `::=` **reachable** *<jml-expression>*`?` **;**

Type checking requirements:

- the optional *<jml-expression>* must be boolean; if not present its default value is **true**.

The `reachable` statement asserts that there exists a feasible execution path that reaches this statement with the given expression true.

Not sure the above is worded correctly - is it that the statement is reachable and whenever it is reached the condition is true?

The terminating semicolon is required and is easily forgotten, since statement is almost always used without the optional expression.

The usual execution of the underlying solver checks whether there are any feasible paths for which an assertions are false. The reachability test is different and typically requires separate executions of underlying solvers, as the test is now for at least one path that reaches the given statement.

The reachable statement is an extension to standard JML.

It has been common practice to insert `assert false;` statements to check whether a given program point is feasible; if it is feasible, the `assert false;` statement will cause a static checking warning. The `reachable` statement accomplished the same purpose with clearer syntax.

10.6 `set` and `debug` statements

Grammar:
<jml-set-statement> `::=` **set** *<java-statement>*
<jml-debug-statement> `::=` **debug** *<java-statement>*

The java-statement in the grammar is not quite right since the statements can include ghost variables.

Type checking requirements:

- the <*java-statement*> may be any single executable Java statement, including a block statement

The DRM requires a set statement to take an assignment expression; the DRM is inconsistent in how it describes debug statements.

A `set` statement marks a statement that is executed during runtime assertion checking or symbolically executed during static checking. As such the statement must be fully executable and may have side effects; also it may contain references and assignments to local ghost variables and ghost fields, and calls of model methods and classes that have executable implementations. The primary motivation for a `set` statement is to assign values to ghost variables, but it can be used to execute any statement.

The semantics of a `debug` statement is the same as the `set` statement except that by default a `debug` statement is ignored. A `debug` statement is only executed when enabled by enabling the optional annotation key `DEBUG` (cf. §**??**). That is, a (single-line, standalone) `debug` statement

`//@ debug` *statement*

is equivalent to

`//+DEBUG@ set` *statement*

This connection between debug and the DEBUG optional key is an extension.

10.7 loop specifications

Grammar:
<*loop-specification*> `::= (` <*loop-clause*> `)*`
<*loop-clause*> `::=` <*loop-invariant*> `|` <*loop-variant*> `|` <*loop-assignable*>

<*loop-invariant*> ::= <*loop-invariant-keyword*> <*jml-expression*> **;**
<*loop-invariant-keyword*> ::= **loop_invariant** | **maintaining**
| **loop_invariant_redundantly**
| **maintaining_redundantly**
<*loop-variant*> ::= <*loop-variant-keyword*> <*jml-expression*> **;**
<*loop-variant-keyword*> ::= **decreases** | **decreasing**
| **decreases_redundantly**
| **decreasing_redundantly**
<*loop-assignable*> ::= <*location-set*> (**,** <*location-set*>)* gterm;

Type checking requirements:

- the <*jml-expression*> in a <*loop-invariant*> must be a boolean expression
- the <*jml-expression*> in a <*loop-variant*> must be a `\bigint` expression *(or just a long?)*
- the <*location-set*>s in a <*loop-assignable*> clause may contain local variables that are in scope at the program location of the loop
- a <*loop-specification*> may only appear immediately prior to a Java loop statement
- the variable scope for the clauses of a <*loop-specification*> includes the declaration statement within a `for` loop, as if the <*loop-specification*> were textually located after the declaration and before the loop body

A special and common case of statement specifications is specifications for loops. In many static checking tools loop specifications, either explicit or inferred, are essential to automatic checks of implementations. *Write this*

10.8 statement (block) specification

Grammar: <*statement-specification*> ::= **refining** <*behavior-seq*>

The semantics of a statement specification are very similar to those of a method specification (cf. §**??**). A method specification states preconditions on the legal states in which a method may be called and gives conditions on what the effects of a method's execution may be, including comparisons between the pre-state (before the method call) and the post-state (after the method completion). Similarly, a statement specification makes assertions about the execution of the statement (possibly a block statement) that follows the statement specification:

- For at least one of the <*behavior*>s in the <*behavior-seq*>, all of the `requires` clauses in that <*behavior*> must be true just prior to the statement being executed
- For any <*behavior*> for which all of the `requires` clauses are true, each other clause must be satisfied.

- The `\result` expression may not be used in any clause

The primary conceptual differences between the method and statement specifications are that

- in the pre- and post-states any local (including ghost) variables that are in scope may be used in the clause expressions
- as there is no return statement, the `\result` expression may not be used

The motivation for a statement specification is that it summarizes the behavior of the subsequent Java statement. The specification should state the behavior of the subsequent statement without reference to implementation details preceding the specification and statement. That is, the preconditions of the statement specification should capture all the requirements necessary to ensure that the statement will satisfy the specifications postconditions, without relying implicit behavior of the statements leading up to the specification and statement.

For example, some block of code may implement a complicated algorithm. The implementation writer may encapsulate that code in a syntactic block and include a specification that describes the effects of the algorithm. Then a tool may separate its static checking task into two parts:

- checking that the implementation in the block does indeed have the effect described by the specification
- checking that the surrounding method satisfies the method's specification when, within its body, the encapsulated block of code is replaced by its specification.

Write this more formally? as a desugaring?

Do we really need the refining keyword?

10.9 `hence_by` statement

Write this Should this be removed?

Chapter 11

JML Expressions

Grammar:

```
<jml-expression> ::=
          «result-expression>
        | «exception-expression>
        | «informal-expression>
        | «old-expression>
        | «key-expression>
        | «lbl-expression>
        | «nonnullelements-expression>
        | «fresh-expression>
        | «type-expression>
        | «typeof-expression>
        | «elemtype-expression>
        | «invariant-for-expression>
        | «is-initialized-expression>
Missing some - check the list
        | «duration-expression>
        | «working-space-expression>
        | «space-expression>
        | ( <jml-expression> )
        | <jml-expression> ? <jml-expression> : <jml-expression>
        | ( + | - | ! | ) <jml-expression>
        | <jml-expression> ( + | - | * | / | % ) <jml-expression>
        | <jml-expression> ( = | != | < | <= | > | >= ) <jml-expression>
        | <jml-expression> ( <==> | <=!=> | ==> | <== | <: ) <jml-expression>
        | <jml-expression> ( <# | <#= ) <jml-expression>
```

shift bit logical dot cast new methodcall ops

Need sections on \count and \values

11.1 Syntax

JML Expressions can include most of the operations defined in Java and additional operations defined only in JML. JML operations are one of four types:

- infix operations that use non-alphanumeric symbols (e.g., `<==>`)
- identifiers that begin with a backslash (e.g., `result`)
- identifiers that begin with a backslash but have a functional form (e.g., `old`)
- methods defined in JML whose syntax is Java-like (e.g., `JML.informal(...)`)

The Java-like forms replicate some of the backslash forms. The backslash forms are traditional JML and more concise. However, the preference for new JML syntax is to use the Java-like form since supporting such syntax requires less modification of JML tools.

11.2 Purity (no side-effects)

Specification expressions must not have side effects. During run-time assertion checking, the execution of specifications may not change the state of the program under test. Even for static checking, the presence of side-effects in specification expressions would complicate their semantics.

11.3 Java operations used in JML

Because of the pure expression rule (cf. §**??**, some Java operators are not permitted in JML expressions:

- allowed: `+ - * / % == != <= >= < > .^ & | && || << >> >>> ?:`
- prohibited: `++ -- = += -= *= /= %= &= |= ^= <<= >>= >>>=`

11.4 Precedence of infix operations

JML infix operators may be mixed with Java operators. The new JML operators have precedences that fit within the usual Java operator precedence order, as shown in Table 11.1.

fix complement operator

Table 11.1: Java and JML precedence (cf. `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html`)

Java operator	JML operator	
highest precedence		associativity
literals, names, parenthesis	quantified	
postfix: `.` `[]` method calls		left
prefix: unary `+ - ! ~` cast new		right
`* / %`		left
binary `+ -`		left
`<< >> >>>`		left
`<= < >= > instanceof`	`<: <# <#=`	left
`== !=`		left
`&`		left
`^`		left
`|`		left
`&&`		left
`||`		left
	`==> <==`	right
	`<==> <=!=>`	left
`?:`		right
assignment, assign-op		right
lowest precedence		

11.5 Well-defined expressions

An expression used in a JML construct must be well-defined, in addition to being syntactically and type-correct. This requirement disallows the use of partial functions with arguments for which the result of the function is undefined. For example, the expression `(x/0) == (x/0)` is considered in JML to be not well-defined (that is, undefined), rather than true by identity. An expression like `(x/y) == (x/y)` (for integer x and y) is true if it can be proved that y is not 0.but undefined if y is possible 0. For example, `y != 0 ==> ((x/y) == (x/y))` is well-defined and true.

The well-definedness rules for JML operators are given in the section describing that operator. The rules for Java operators are given here. They presume that the expressions are type correct.

- literals and names are well-defined
- parenthesis operator: well-defined iff the operand is well-defined
- member-of (dot operator): well-defined iff the expression to the left of the dot is well-defined and not null
- array element ([] operator): well-defined iff the array expression is well-defined and not null and the index expression is well-defined and within range
- method calls: well-defined iff (a) the receiver and all arguments are well-defined and (b) if the method is not static, the receiver is not null and (c) the method's precondition and invariants are true and (d) the method does not throw any Exceptions
- cast operator: well-defined iff the argument is well-defined and, for limited range integral types, the result is not out of range for the type
- implicit unboxing (conversion from object to primitive): well-defined if the argument is well-defined and not null
- new operator: well-defined iff (a) all arguments to the constructor call are well-defined, (b) the preconditions and static invariants of the constructor are satisfied by the argument, (c) and the constructor does not throw any Exceptions
- bit operators and non-short-circuit boolean operators (`& | ^ ~`): well-defined iff all operands are well-defined
- shift operators (`<< >> >>>`): well-defined iff all operands are well-defined. Note that Java defines the shift operations for any value of the right-hand operand; the value is trimmed to 5 or 6 bits by a modulo operation appropriate to the bit-width of the left-hand operand. JML tools may choose to raise a warning if the value of the right-hand operand is outside the 'expected' range.
- divide and modulo operators: well-defined iff both operands are well-defined and the divisor is not zero and, for limited range integral types, the result is not out of range for the type
- other arithmetic operators: well-defined iff both operands are well-defined and, for limited range integral types, the result is not out of range for the type
- string concatenation: well-defined if the operands are well-defined (they may be null)
- boolean negation operator (`!`): well-defined iff the operand is well-defined

floating point operations?

- relational and equality operators: well-defined iff both operands are well-defined
- short-circuit boolean and operator (`&&`): well-defined iff
  - the left operand is well-defined and
  - either the left operand is false or the right-operand is well-defined
- short-circuit boolean or operator (`||`): well-defined iff
  - the left operand is well-defined and
  - either the left operand is true or the right-operand is well-defined
- ternary operator (`?:`): well-defined iff
  - the condition is well-defined and
  - the then operand is well-defined whenever the condition is true and
  - the else operand is well-defined whenever the condition is false.

  For example `(o != null ? o.x : 0)` is well-defined (if it is type-correct) because (a) the condition `o != null` is well-defined and (b) the then expression `o.x` is well-defined if the condition `o != null` is true and (c) the else expression is always well-defined.

11.6 org.jmlspecs.lang.JML

Say more

11.7 Implies and reverse implies: `==>` `<==`

Type information:

- two arguments, each an expression of boolean type
- a ==> expression is well-defined if the left operand is well-defined and either the left operand is false or the right operand is well-defined
- a <== expression is well-defined if the left operand is well-defined and either the left operand is true or the right operand is well-defined
- result is boolean

The ==> operator denotes implication and is a short-circuit operator. It is true if the left-hand operand is false or the right-hand operand is true; if the left operand is false, the right operand is not evaluated and may be undefined. The operation

<*left*> ==> <*right*>

is equivalent to

(**!** <*left*>) || <*right*> .

The <== operator denotes reverse implication and is a short-circuit operator. It is true if the left-hand operand is true or the right-hand operand is false; if the left operand is true, the right operand is not evaluated and may be undefined. The operation

<*left*> <== <*right*>

is equivalent to

<*left*> || (**!** <*right*>) .

Both operators are right associative. For example `P ==> Q ==> R` is parenthesized as `P ==> (Q ==> R)`. This is the natural association from logic: `(P ==> Q) ==> R` is equivalent to `(P && !Q) || R` whereas `P ==> (Q ==> R)` is equivalent to `!P || !Q || R`.

Note that because of the short-circuit nature of these two operators <*left*> ==> <*right*> is not necessarily equivalent to <*right*> <==<*left*>. In particular, if <*right*> is undefined then <*left*> ==> <*right*> is either true or undefined, whereas <*right*> <== <*left*> is always undefined.

11.8 Equivalence and inequivalence: <==> <=!=>

Type information:

- two arguments, each an expression of boolean type
- the expression is well-defined if both operands must be well-defined
- result is boolean

The <==> operator denotes equivalence: it is true if both operands are true or both are false. It is equivalent to equality (==), except that it is lower precedence. For example, `P && Q <==> R || S` is `(P && Q) <==> (R || S)`, whereas `P && Q == R || S` is `(P && (Q == R)) || S`.

The <=!=> operator denotes inequivalence: it is true if one operand is true and the other false. It is equivalent to inequality (!=), except that it is lower precedence. For example, `P && Q <=!=> R || S` is `(P && Q) <=!=> (R || S)`, whereas `P && Q != R || S` is `(P && (Q != R)) || S`.

Both of these operators are associative and commutative. Accordingly left- and right-associativity are equivalent. The operators are not chained: `P <==> Q <==> R` is `(P <==> Q) <==> R`, not `(P <==> Q) && (Q <==> R)`; for example, `P <==> Q <==> R` is true if P is true and Q and R are false. Similarly `P <=!=> Q <=!=> R` is `(P <=!=> Q) <=!=> R` and is true if P is true and Q and R are false.

11.9 JML subtype: <:

Type information:

- two arguments, each of type `\TYPE`
- both operands must be well-defined
- result is boolean

The <: operator denotes JML subtyping: the result is true if the left operand is a subtype of the right operand. Note that the argument types are
`TYPE`, that is JML types (cf. §**??**). *Say more about relationship to Java subtyping*

Note that the operator would be better named `<:=`, since it is true if the two operands are the same type.

11.10 Lock ordering: `<# <#=`

Type information:

- two arguments, each of reference type
- both operands must be well-defined and both must not be null
- result is boolean

It is useful to establish an ordering of locks. If lock A is always acquired before lock B (when both locks are needed) then the system cannot deadlock by having one thread own A and ask for B while another thread holds B and is requesting A. Specifications may specify an intended ordering using axioms and then check that the ordering is adhered to in preconditions or assert statements. Neither Java nor JML defines any ordering on locks; the lock ordering operator enables the specifier to write appropriate statements about the desired ordering.

The `<#` operator is the 'less-than' operator on locks; `<#=` is the 'less-than-or-equal' version. That is

```
a <#= b === ( a <# b | a == b )
```

Previously in JML, the lock ordering operators were just the `<` and `<=` comparison operators. However, with the advent of auto-boxing and unboxing (implicit conversion between primitive types and reference types) these operators become ambiguous. For example, if a and b are `Integer` values, then $a < b$ could have been either a lock-ordering comparison or an integer comparison after unboxing a and b. Since the lock ordering is only a JML operator and not Java operator, the semantics of the comparison could be different in JML and Java. To avoid this ambiguity, the syntax of the lock ordering operator was changed and the old form deprecated.

11.11 `\result`

Grammar:
<result-expression> `::=` **\result**

Type information:

- no arguments
- always well-defined, if type-correct
- result type is the return type of the method in whose specification the expression appears
- may only be used in `ensures`, `duration`, and `working_space` clauses

The `\result` expression denotes the value returned by a method. The expression is only permitted in `ensures` clauses of the method's specification. It is a type-error to use `\result` in the specification of a constructor or a method whose return type is `void`.

11.12 `\exception`

`\exception` is an OpenJML extension

Grammar:
<exception-expression> ::= **\exception**

Type information:

- no arguments
- always well-defined, if type-correct
- result type is the type of the exception given in the `signals` clause
- only permitted in the `signals`, `duration`, and `working_space` clauses

The `\exception` expression denotes the exception object within a `signals` clause. The expression is only permitted in `signals` clause of the method's specification. It is an alternative form to using a variable declared in the `signals` clauses's declaration. For example, the following two constructions are equivalent:

```
//@ signals (RuntimeException e) ...  e ...  ;
//@ signals (RuntimeException) ...  \exception ...  ;
```

11.13 `\old`, `\pre`, and `\past`

Grammar:
<old-expression> ::=
(**\old(** *<jml-expression>* (, *<label>*)?)
(**\pre(** *<jml-expression>*)
(**\past(** *<jml-expression>*)
<label> ::= *<id>*

Any pre-defined labels?

`\past` is an extension

Text needed

11.14 \key

The \key expression is an OpenJML extension

Grammar:
<key-expression> ::= (**\key(** *<string-literal>*)

Type information:

- The argument must be a compile-time string literal
- The result is a boolean

This expression enables a JML expression to vary depending on the settings of optional, tool-specified keys. These are the same keys as are used for optional annotation comments (cf. §**??**). The \key expression is translated to a boolean literal true or false depending on whether the key is defined or not. Some tools may subsequently use constant folding to avoid processing or executing unreachable parts of expressions.

Example: The expression

!\key("RAC") ==> state == 0

might be used in a situation where `state` is a model field that is not compiled in RAC mode. This expression is equivalent to

- `true` if `RAC` is defined (so that `\key("RAC")` is true)
- `state == 0` if `RAC` is not defined (so that `\key("RAC")` is false)

Mechanisms for defining keys are provided by tools and are not defined in JML itself.

11.15 \lblpos, \lblneg, \lbl, and JML.lblpos(), JML.lblneg(), JML.lbl()

\lbl and the JML forms are extensions

Grammar:

<lbl-expression> ::=
(**\lbl(** *<id>* , *<jml-expression>*)
| (**\lblpos(** *<id>* , *<jml-expression>*)
| **\lblneg(** *<id>* , *<jml-expression>*)

Type information: `\lblpos`, `\lblneg`

- special syntax, including a boolean expression
- well-defined if the boolean expression is well-defined
- result type is boolean; value is the same as the expression in the argument

Type information: `JML.lblpos()`, `JML.lblneg()`

- first argument is a String literal, the second has boolean type
- well-defined if the arguments are well-defined
- result type is boolean; value is the second argument

Type information: `\lbl`

- special syntax, including an expression
- well-defined if the argument expression is well-defined
- result type is the same as the expression; value is the value of the expression

Type information: `JML.lbl()`

- first argument is a String literal, the second has any type
- well-defined if the arguments are well-defined
- result type is the same as the type of the second argument; value is the second argument

These expressions are used for debugging. When static checking finds that some assertion is invalid and generates a counterexample for that invalid assertion, then the value of the expression in contained in the lbl expression is reported as part of the counterexample. When runtime assertion checking is used, these expressions print the String id and the value of expression. In each case, the construct simply passes on the type and value of its expression, possibly generating some debug output in the process.

The `\lblpos`, `\lblneg`, and `\lbl` expressions have a non-standard syntax:

(\lbl *<id>* *<expression>*)

with no comma between what would be the arguments. Here the *<id>* is a Java identifier. The `JML.lbl` forms are standard functional forms: the first argument is a String literal; the second is an expression. A String literal is required instead of a String expression because the value is used to identify the output as coming from this expression and it is not evaluated during static checking.

In the positive and negative forms, the argument is a boolean expression. Output is generated by lblpos only if the expression is true; output is generated by lblneg only if the expression is false. In the neutral form (`\lbl` and `JML.lbl`), output is generated whatever the value. For any type, the value is converted to a String value as is customary in Java (e.g., using toString()).

Examples: *Examples needed*

11.16 \nonnullelements

Grammar:

<nonnullelements-expression> ::=
(**\nonnullelements**(*<jml-expression>* (, *<jml-expression>*)*
)

(**\nonnullelements**( )

Type information:

- strict JML: one argument, an expression of array type, may be null
- OpenJML extensions: any number of arguments, each an expression of array type
- always well-defined, if type-correct
- result type is boolean

The argument of the `\nonnullelements` expression must be an expression that evaluates to an array. The `\nonnullelements` expression is true if the argument is true and each element of the array (if any) is not null.

If there are multiple arguments, then the result is true if all of the arguments are non-null and have all non-null array elements. If `\nonnullelements` has no arguments, its value is true.

Perhaps allow the argument to be reference type - result is true if the argument is non-null and, if an array, all elements are non-null. Are elements non-null recursively?

Allow any number of arguments?

11.17 `\fresh`

Grammar:

<fresh-expression> ::=
(**\fresh**(*<jml-expression>* (, *<jml-expression>*)*)
(**\fresh**( )

Type information:

- strict JML: one argument, an expression of reference type
- OpenJML extensions: any number of arguments, each an expression of reference type
- well-defined, if type-correct and all arguments are non-null
- result type is boolean
- use: `\fresh` may be used only in `ensures` and `signals` clauses

The argument of the `\fresh` expression must be an expression that evaluates to a non-null reference. The `\fresh` expression is true if the argument is a reference that was not allocated in the pre-state.

If there are multiple arguments, then the result is true if each argument is itself `\fresh`; that is, the value with n arguments is the conjunction of n terms each of which is `\fresh` applied to one of the arguments in turn. If `\fresh` has no arguments, its value is true.

Standardize the JML extension?

11.18 informal expression: `(*...*)` and `JML.informal()`

Grammar:
<informal-expression> ::=
((* *FIXME* *)

Type information:

- one argument
- always well-defined, if type-correct
- result type is boolean; value is always true

The syntax of the informal expression is

`(* ... *)`,

where the ... denotes any sequence of characters not including line breaks or the two-character sequence `*)`. An alternate form is

`JML.informal(`*<expression>*`)` ,

where *<expression>* is a String expression, though typically a String literal. The character sequence and the string expression are natural language text that may be ignored by JML tools; the intent is to convey to the reader some natural language specification that will not be checked by automated tools.

In the second form, the argument is type checked and must have type `java.lang.String`; it is not evaluated. It is generally a string literal.

The expression always has the value true.

Examples:

```
//@ ensures (* data structure is self-consistent *);
//@ ensures JML.informal("data structure is self-consistent");
public void m()  ...
```

11.19 \type

Grammar:
<type-expression> ::=
(**type**(*<jml-type-expression>*)

Type information:

- one argument, a type name
- always well-defined, if type-correct
- result type is \TYPE

This expression is a type literal. The argument is the name of a type as might be used in a declaration; the type may be a primitive type, a non-generic reference type, a generic type with type arguments or an array type. The value of the expression is the JML type value corresponding to the given type. It is analogous to .class in Java, which converts a type name to a value of type Class. The type name is resolved like any other type name, with respect to whatever type names are in scope.

Generic types must be fully parameterized; no wild card desingations are permitted.

What about type variables

For more discussion of JML types and their relationships to Java type, see §**??**.

Examples:

```
//@ ...  \type(int) ...
//@ ...  \type(Integer) ...
//@ ...  \type(java.lang.Integer) ...
//@ ...  \type(java.util.LinkedList<String>) ...
//@ ...  \type(java.util.LinkedList<String>[]) ...
```

11.20 \typeof

Grammar:
<typeof-expression> ::=
(**typeof**(*<jml-expression>*)

Type information:

- one expression argument, of any type
- well-defined if the argument is well-defined and not null
- result type is \TYPE

The \typeof expression returns the dynamic type of the expression that is its argument. In run-time checking this may require evaluating the argument. This operation returns a JML type (\typeof); it is analogous to the Java method .getClass(), which returns a Java type value (of type Class).

Verify that primitive types are allowed

Examples:

```
Object o = new Integer(5);
// o has static type Object, but dynamic type Integer
//@ assert \typeof(o) == \type(Integer); // - true
//@ assert \typeof(o) == \type(Object); // - false
//@ assert \typeof(5) == \type(int); // - true
```

11.21 \elemtype

Grammar:

<elemtype-expression> ::=
(**\elemtype(** *<jml-expression>*)

Type information:

- one argument, of type \TYPE
- well-defined iff the argument is well-defined and the value is an array type
- result type is \TYPE

This operator returns the element type of an array type.

What if the argument is not an array type? Should we allow a null value of \TYPE

Examples:

```
//@ assert \elemtype(\type(int[])) == \type(int);
//@ assert \elemtype(\type(int)) == \type(int); // - undefined
```

11.22 \is_initialized

Grammar:

<is-initialized-expression> ::=
(**\is_initialized(** *<jml-expression>* (, *<jml-expression>*)*)
| (**\is_initialized(**)

Text needed

11.23 \invariant_for

Grammar:
<invariant-for-expression> ::=
(**\invariant_for(** *<jml-expression>* (**,** *<jml-expression>*)*
) | (**\invariant_for( )** *Allow expr or type in grammar*

Type information:

- Strict JML: one argument, of type `Object` or a typename, but not a primitive expression
- Extension: any number of arguments
- well-defined iff the argument is well-defined and the value is a reference type but not an array type
- result type is `boolean`

This expression with one expression argument is equivalent to the conjunction (with `&&`) of the static and non-static JML-visible invariants in the static type of the receiver and all its super classes and interfaces (recursively), with the argument as the receiver for the invariants.

OpenJML allows as an extension one typename argument. The expression with a typename argument is equivalent to the conjunction (with `&&`) of the *static* JML-visible invariants in the named type and all its super classes and interfaces (recursively).

In each case the order of invariants is (1) that invariants of super classes and interfaces occur before derived classes and interfaces, (2) Object is first and the named type is last, and (3) within a type, invariants occur in textual order.

Strict JML requires that there be just one argument. As an extension, OpenJML allows any number of arguments. The expression is then equivalent to the conjunction of the values for each argument, in order, conjoined by the short-circuit operator `&&`. When there are no arguments, the value of `\invariant_for()` is `true`.

Are the invariants of superclasses and interfaces included?

Are both static and non-static invariants included?

The DRAFT JML reference manual does not mention JML-visibility, but I presume that must be the case.

multiple arguments, typename argument

The DJMLRM does not mention a static-only version of \invariant_for - so this is an extension?

11.24 \not_modified

Grammar:

<not-modified-expression> ::=
(**\not_modified(** *<jml-expression>* (, *<jml-expression>*)*) | (**\not_modified(**)

Type information:

- Strict JML: one argument, an expression of any type other than void
- Extension: any number of arguments, each expression of any type other than void
- well-defined iff the arguments are well-defined
- result type is `boolean`
- use: only in `ensures` or `signals` clauses

A `\not_modified` expression is a two-state expression that may occur only in `ensures` or `signals` clauses. It satisfies this equivalence:

`\not_modified(`*o*`) == ( \old(`*o*`) == (` *o* `) ) )`

The argument may be null.

A `\not_modified` expression with multiple arguments is the conjunction of the corresponding terms each with one argument; if `\not_modified` has no arguments, its value is true.

The RM says the argument is a store-ref list, rather than an expression. Which do we want? A store-ref-list allows constructions such as o. or a[*] or a[1..6] but not a+b.*

11.25 \lockset and \max

Text needed

11.26 \reach

Text needed

11.27 \duration

Grammar:
<duration-expression> ::=
(**duration**(*<jml-expression>*)

Type information:

- one argument, an expression of any type
- well-defined
- result type is `long`

Here we say that the argument is an expression, whereas the DRM says it must be an explicit method or constructor call.

The value of a `\duration` expression is the maximum number of virtual machine cycles needed to evaluate the argument. The argument is not actually executed and need not be pure. However, reasoning about assertions containing `\duration` expressions is based on the specifications of method calls within the expression, not on their implementation. Consequently, for a `\duration` expression to be useful, any methods or constructors within its argument must have a `duration` expression as part of its method specification.

The argument must be an executable expression because different expressions (e.g., method calls with different arguments) may consume different numbers of machine cycles during execution.

Say more about what a virtual machine cycle is.

11.28 \working_space

Grammar:
<working-space-expression> ::=
(**working_space**(*<jml-expression>*)

Type information:

- one argument, of any type
- well-defined
- result type is `long`

Here we allow the argument to be any expression. The DRM requires the argument to be a method or constructor call.

The result of the `\working_space` expression is the number of bytes of heap space that would be required to evaluate the argument, if it were executed. The argument is not actually executed and may contain side-effects. That is, if `\working_space(`*expr*`)` free bytes are available in the system, then evaluating *expr* will not cause an OutOfMemory error. *Is this last sentence true?*

The argument must be an executable expression because different expressions (e.g., method calls with different arguments) may consume different amounts of memory space during execution.

11.29 `\space`

Grammar:

<space-expression> ::=
(**\space(** *<jml-expression>* **)**

Type information:

- one argument, of any reference type
- well-defined iff the argument is well-defined *Must the argument be well-defined?*
- result type is `long`

The result of a `\space` expression is the number of bytes of heap space occupied by the argument. This is a shallow measure of space: it does not include the space required by objects that are referred to by members of the object, just the space to hold the references themselves and any primitive values that are members of the argument.

not-assigned, not-modified, only-accessed, only-called, only-assigned, only-captured, spec-quantified-expr Text needed

Chapter 12

Arithmetic modes

12.1 Description of arithmetic modes

Programming languages use integral and floating-point values of various ranges and precisions. However, often specifications are written and understood as mathematical integer and real values. Chalin [**?**] surveyed programmer expectations and desires and identified three useful arithmetic modes:

- Java mode: values belong to one of Java's fixed-bit-length data types; overflows and underflows either occur silently or result in undefined values according to the rules of Java arithmetic
- Safe mode: values belong to one of Java's fixed-bit-length data types; overflows and underflows cause static or dynamic warnings
- Math mode: numeric values are promoted to mathematical types prior to arithmetic operations; warnings may be issued when values are stored back into fixed-bit-length variables (in the description below, this mode is called 'bigint' mode, but is the mathematical mode for both integers and reals).

Chalin proposed that most of the time, programmers would like Safe mode semantics for programming language operations and Math mode for specification expressions.

Question: In math mode is it just the operations that are on math types and then casts or writes to variables might trigger warnings; or are all integral data types implicitly bigint and real?

In JML, the type of mathematical integers is expressed as `\bigint` and the type of mathematical reals is expressed as `\real`. Static checking can reason about these types using usual logics with arithmetic; runtime checking uses `java.math.BigInteger` to represent `\bigint` and `ixorg.jmlspecs.openjml.Real` to represent `\real`.

JML contains a number of modifiers and pseudo-functions to control which mode is operational for a given sub-expression. As would be expected, the innermost mode indicator in scope for a given expression overrides enclosing arithmetic mode indicators. The arithmetic mode can be set separately for the Java source code and the JML specifications.

- the class and method modifiers `code_java_math`, `code_safe_math`, and `code_bigint_math`, and corresponding annotation types `CodeJavaMath`, `CodeSafeMath`, and `CodeBigintMath`, set the default arithmetic mode for all expressions in Java source code within the class or method (unless overridden by a nested mode indicator).
- the class and method modifiers `spec_java_math`, `spec_safe_math`, and `spec_bigint_math`, and corresponding annotation types `SpecJavaMath`, `SpecSafeMath`, and `SpecBigintMath`, set the default arithmetic mode to be used within JML specifications, within the respective class or method.
- the operators `\java_math`, `\safe_math`, and `\bigint_math` take one argument, an expression, and set the arithmetic mode for evaluating that expression (unless overridden by a nested arithmetic mode operator); the result of these operators has the type and value of its argument, adjusted for the arithmetic mode.
- the default arithmetic mode for the whole static or dynamic analysis can be set using command-line options (or in the GUI preferences): `-code-math` and `-spec-math`, which can have the values `java`, `safe`, and `bigint`.
- the default settings of the global options are `-code-math=java` and `-spec-math=bigint`.

Change the annotations to be simply @CodeMath and @SpecMath with a value?

The math mode affects the semantics of these operators:

- arithmetic: unary plus and minus and binary `+ - * / %`
- shift operations: `« » »>`
- cast operation
- *Math functions ???*

The semantics of these operations in each mode are described in the following sections.

Say more about the explicit semantics

12.2 Semantics of Java math mode

Java defines several fixed-precision integral and floating-point data types. In addition JML allows the `\bigint` and `\real` data types. The arithmetic and shift operators act on these data types as follows:

- implicit conversion. The operands are individually converted to potentially larger data types as follows:
  - if either operand is `\real`, the other is converted to `\real`, else
  - if one operand is `\bigint` and the other either `double` or `float`, they both are converted to `\real`, else
  - if either operand is `double`, the other is converted to `double`, else
  - if either operand is `float`, the other is converted to `float`, else
  - if either operand is `\bigint`, the other is converted to `\bigint`, else
  - if either operand is `long`, the other is converted to `long`, else
  - both operands are converted to `int`.
- the result type of each arithmetic operator is the same as that of its implicitly converted operands
- the result type of a shift operator is the same as its left-hand operand
- `double` and `float` operators behave as defined by the IEEE standard
- the unary plus operation simply returns its operand
- the unary minus operation, when applied to the least `int` or `long` value will overflow, returning the value of the operand
- binary add, subtract, and multiply operations on `int` or `long` values may overflow or underflow; the result is truncated to the number of bits of the result type
- the binary divide operation will overflow when the least value of the type is divided by -1. The result is the least value of the result type.
- the binary modulo operation does not overflow. Note that the sign of the result is the same as the sign of the *dividend*, and that it is always true that $x == (x/y) * y + (x\%y)$ for x and y both int or both long.[1]
- the shift operators apply only to integral values. Note that in Java, $x << y == x << (y\&n)$ where n is 31 when x is an `int` and 63 if x is a `long`. However, no such adjustment to the shift amount happens when the type is `\bigint`.
- In narrowing cast operations, the value of the operand is truncated to the number of bits of the given type.

12.3 Semantics of Safe math mode

The result of an operation in safe math mode is the same as in Java math mode, except that any out of range value causes a warning in static or dynamic checking. These

[1] `https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.17.3`

warnings are produced in these cases:

- arithmetic operations on fixed-bit-width integral types produce results that overflow or underflow; no warnings are produced for floating point or mathematical types
-

12.4 Semantics of Bigint math mode

12.5 Real arithmetic and non-finite values

TBD -reference what is done in ACSL

Chapter 13

Universe types

Chapter 14

Model Programs

Describe the intent, syntax and semantics of model programs

Chapter 15

Obsolete and Deprecated Syntax

describe deprecated syntax, obsolete syntax, incompatible changes from past versions

Appendix A

Summary of Modifiers

The tables on the following pages summarizes where the various Java and JML modifiers may be used.

Fix up page break Review for correctness and completeness.

Missing: non_null_by_default

Add in secret, query

Grammatical construct	Java modifiers	JML modifiers
All modifiers	`public protected private abstract static final synchronized transient volatile native strictfp`	`spec_public spec_protected model ghost pure instance helper non_null nullable nullable_by_default monitored uninitialized`
Class declaration	`public final abstract strictfp`	`pure model nullable_by_default spec_public spec_protected`
Interface declaration	`public strictfp`	`pure model nullable_by_default spec_public spec_protected`
Nested Class declaration	`public protected private static final abstract strictfp`	`spec_public spec_protected model pure`
Nested interface declaration	`public protected private static strictfp`	`spec_public spec_protected model pure`
Local Class (and local model class) declaration	`final abstract strictfp`	`pure model`

Grammatical construct	Java modifiers	JML modifiers
Type specification (e.g. invariant)	`public protected private static`	`instance`
Field declaration	`public protected private final volatile transient static`	`spec_public spec_protected non_null nullable instance monitored`
Ghost Field declaration	`public protected private static final`	`non_null nullable instance monitored`
Model Field declaration	`public protected private static`	`non_null nullable instance`
Method declaration in a class	`public protected private abstract final static synchronized native strictfp`	`spec_public spec_protected pure non_null nullable helper extract`
Method declaration in an interface	`public abstract`	`spec_public spec_protected pure non_null nullable helper`
Constructor declaration	`public protected private`	`spec_public spec_protected helper pure extract`
Model method (in a class or interface)	`public protected private abstract static final synchronized strictfp`	`pure non_null nullable helper extract`
Model constructor	`public protected private`	`pure helper extract`
Java initialization block	`static`	-
JML `initializer` and `static_initializer` annotation	-	-
Formal parameter	`final`	`non_null nullable`
Local variable and local ghost variable declaration	`final`	`ghost non_null nullable uninitialized`

Appendix B

Grammar Summary

Automatic collection of all of the grammar productions listed elsewhere in the document

Appendix C

Type Checking Summary

This was in the DRM outline - is there something to be put in here? If it is to be collected from the rest of the document, we need to place markers to identify the relevant stuff.

Appendix D

Verification Logic Summary

This was in the DRM outline. What was its intent? Is it the same as a section on semantics and translation?

Appendix E

Differences in JML among tools

Some material is in the DRM. Needs to be enhanced. SHould have a detailed comparison with ACSL, for example – see the appendix of the ACSL documentation.

Appendix F

Misc stuff to move, incorporate or delete

F.0.1 Finding specification files and the refine statement

JML allows specifications to be placed directly in the .java files that contain the implementation of methods and classes. Indeed, specifications such as assert statements or loop invariants are necessarily placed directly in a method body. Other specifications, such as class invariants and method pre- and post-conditions, may be placed in auxiliary files. For classes which are only present as .class files and not as .java files, the auxiliary file is a necessity.

Current JML allows one such auxiliary file per `.java` file or corresponding `.class` file. It is similar to the corresponding `.java` file except that

- it has a `.jml` suffix
- it contains no method bodies (method declarations are terminated with semicolons, as if they were abstract)
- TBD - field initializations?

The `.jml` file must be in the same package as the corresponding `.java` file and has the same name, except for the suffix. It need not be in the same folder, though the tail of the path to the folder containing the `.jml` file must still correspond to the package containing the `.java` and `.jml` files. If there is no source file, then there is a .jml file for each compilation unit that has a specification. All the nested, inner, or top-level classes that are defined in one Java compilation unit will have their specifications in one corresponding `.jml` file.

The search for specification files is analogous to the way in which `.class` files are found on the *classpath*, except that the *specspath* is used instead. To find the specifications for a public top-level class *T*:

- look in each element of the *specspath* (cf. section TBD), in order, for a fully-qualified file whose name is *T*.`jml`. If found, the contents of that file are used as the specifications of *T*.
- if no such `.jml` file is found, look in each element of the *specspath*, in order, for a fully-qualified file whose name is *T*.`java`.

There are two (silent) consequences of this search algorithm that can be confusing:

- If both a `.jml` and a `.java` file exist on the specspath and both contain JML specification text, the specifications in the `.java` file will be (silently) ignored.
- If a `.java` file is listed on the command-line it will be compiled (for its Java content), but if it is not a member of an element of the *specspath*, it will (silently) not be used as the source of specifications for itself.

Obsolete syntax. The `refine` and `refines` statements are no longer recognized. The previous (complicated) method of finding specification files and merging the specifications from multiple files is also no longer implemented. The only specification file suffix allowed is `.jml`; the others — `.spec`, `.refines-java`, `.refines-spec`, `.refines-jml` — are no longer implemented.

In addition, the `.jml` file is now sought before seeking the `.java` file; if a `.jml` file is found anywhere in the specs path, then any specifications in the `.java` file are ignored. This is a different search algorithm than was previously used.

F.0.2 Model import statements

This section will be added later. Java `import` statements introduce class names into the namespace of a .java file. JML has a `model import` statement:

```
//@ model import ...
```

The effect of a JML model import statement is the same as a Java import statement, except that the names imported by the JML statement are only visible within JML annotations. If the model import statement is within a .jml file, the imported names are visible only within annotations in the .jml file, and not outside JML annotations and not in the .java file.

Note: Most tools only approximately implement this feature. For example, see FIXME for a discussion of this feature in OpenJML.

F.0.3 Modifiers

Modifiers are JML keywords that specify JML characteristics of methods, classes, fields, or variables. They are syntactically placed just like Java modifiers, such as `public`.

JML Keyword	Java annotation	class	interface	method	field declaration	variable declaration
code	Code			X		
code_bigint_math	CodeBigintMath					
code_java_math	CodeJavaMath					
code_safe_math	CodeSafeMath					
extract	Extract					
ghost	Ghost				X	X
helper	Helper			X		
instance	Instance					
model	Model					
monitored	Monitored					
non_null	NonNull			X	X	X
non_null_by_default	NonNullByDefault	X	X	X		
nullable	Nullable			X	X	X
nullable_by_default	NullableByDefault	X	X	X		
peer	Peer					
pure	Pure	X	X	X		
query	Query					
readonly	Readonly					
rep	Rep					
secret	Secret					
spec_bigint_math	SpecBigintMath					
spec_java_math	SpecJavaMath					
spec_protected	SpecProtected					
spec_public	SpecPublic					
spec_safe_math	SpecSafeMath					
static	Static					
uninitialized	Uninitialized					

Table F.1: Summary of JML modifiers. All Java annotations are in the `org.jmlspecs.annotation` package.

Reuse this table?

CHeck the table; add section references; add where allowed; indicate which are type modifiers; turn headings 90 degrees.

Obsolete syntax. JML no longer defines the modifier `weakly`.

JML defines some statements that are used in the body of a method's implementation. These are not method specifications per se; rather, they are assertions or assumptions that are used to aid the proof of the specifications themselves, in the way that lemmas are aids to proving a resulting theorem. They can also be used to state predicates that the user believes to be true, and wants checked, or assumptions that are true but are too difficult for the prover to prove itself.

F.0.4 JML expressions

Expressions in JML annotations are Java expressions with three adjustments:

- Expressions with side-effects are not allowed. Specifically, JML excludes
  - the ++ and – pre- and post- increment and decrement operations
  - the assignment operator
  - assignment operators that combine an operation with assignment (e.g., +=)
  - method invocations that are not explicitly declared `pure` (cf. §TBD)
- JML adds additional operators to the Java set of operators, discussed in subsection §**??** below.
- JML adds specific keywords that are used as constants or function-like expressions within JML expressions, discussed in subsection §**??** below

JML lock ordering operators (`<#`) and `<#=`) The lock ordering operators are used to determine ordering among objects used for locking in a multi-threaded application; the operands are any Java objects. The only predefined property of these operators is that for any two object references `o` and `oo`, `o <#= oo` is equivalent to `o == oo || o <# oo`; that is `<#` is like less than and `<#=` is like less-than-or-equals. There is no predefined ordering among objects. The user must define an intended ordering with some axioms or invariants. An example of using the lock ordering operators for specification and reasoning about concurrency is found in §**??**.

TBD - add ++ – into the table as Java only; check precedence

F.0.5 Code contracts

This section will be added later.

F.1 JML modifiers and Java annotations

The Java Modeling Language was defined prior to the introduction of annotations in Java. Some, but not all, of the features of JML can now be textually represented as Java annotations. Currently JML supports both the old and new syntactic forms.

Modifiers

Modifiers are JML keywords that specify JML characteristics of methods, classes, fields, or variables. Examples are `pure`, `model`, and `ghost`. They are syntactically placed just like Java modifiers, such as `public`.

`new () [] .` and method calls		
unary + unary - `!` (typecast)	-	
`* / %`	L	
+ (binary) - (binary)	L	
`« » »>`	L	
`< <= > >= <: instanceof <# <#=`	-	`<:` is the JML subtype operation (§**??**); <# and <#= are lock ordering operators (§F.0.4)
`== !=`	L	
`&`	L	
`^`	L	
`|`	L	
`&&`	L	
`||`	L	
`==> <==`	?	JML implies and reverse implies (§**??**)
`<==> <=!=>`	?	JML equivalence and inequivalence (§**??**)
`?:`	-	
`= *= /= %= += -= «= »= »>= &= ^ |=`	L	Java only

Table F.2: Java and JML operators, in order of precedence, from highest (most tightly binding) to lowest precedence. Operators on the same line have the same precedence. The associativity is given in the central column.

Each such modifier has an equivalent Java annotation. For example

```
/*@ pure */ public int m(int i)  ...
```

can be written equivalently as

```
org.jmlspecs.annotation.Pure public int m(int i)  ...
```

The `org.jmlspecs.annotation` prefix can be made implicit in the usual way by including the import statement

```
import org.jmlspecs.annotation.Pure;
```

Note that in the second form, the `pure` designation is now part of the *Java* program and so the import of the `org.jmlspecs.annotation` package must also be in the Java program, and the package must be available to the Java compiler.

All of the modifiers, their corresponding Java annotations, and the locations in which they may be used are described in §F.3.5.

Type modifiers

Some modifiers are actually type modifiers. In particular `non_null` and `nullable` are in this category. Thus the description of the previous subsection (§F.1) apply to these as well.

However, Java 1.8 allows Java annotations to be applied to types wherever type names may appear. For example

```
(@NonNull String)toUpper(s)
```

is allowed in Java 1.8 but is forbidden in Java 1.7.

Need additional discussion of the change in JML for Java 1.8, especially for arrays.

Method specification clauses

This section will be added later.

Class specification clauses

This section will be added later.

Statement specifications

JML specifications that are statements within the body of a method have no equivalent as Java annotations. These include loop specifications, assert and assume statements, ghost declarations, set and debug statements, and specifications on individual statements.

F.2 org.jmlspecs.lang package

Some JML features are defined in the `org.jmlspecs.lang` package. The `org.jmlspecs.lang` package is included as a model import by default, just as the `java.lang` package is included by default in a Java file. `org.jmlspecs.lang.*` contains these elements:

- `JML.informal(`*`<string>`*`)` : This method is a replacement for (and is equivalent to) the informal expression syntax (§**??**) `(* ... *)`. Both expressions return a boolean value, which is always `true`.
- TBD

The definition of the Java Modeling Language is contained in the JML reference manual.[**?**] This document does not repeat that definition in detail. However, the following sections summarize the features of JML, indicate what is and is not implemented in OpenJML, describes any extensions to JML contained in OpenJML, and includes comments about relevant implementation aspects of OpenJML.

F.3 JML Syntax

F.3.1 Syntax of JML specifications

JML specifications may be written as Java annotations. Currently these are only implemented for modifiers (cf. section TBD). In Java 8, the use of Java annotations for JML features will be expanded.

JML specifications may also be written in specially formatted Java comments: a JML specification includes everything between either (a) an opening `/*@` and closing `*/` or (b) an opening `//@` and the next line ending character (`\n` or `\r`) that is not within a string or character literal.

Such comments that occur within the body of a class or interface definition are considered to be a specification of the class, a field, or a method, depending on the kind of specification clause it is. JML specifications may also occur in the body of a method.

Obsolete syntax. In previous versions of JML, JML specifications could be placed within javadoc comments. Such specifications are no longer standard JML and are not supported by OpenJML.

F.3.2 Conditional JML specifications

ofJML has a mechanism for conditional specifications, based on a system of keys. A key is an identifier (consisting of ASCII alphanumeric and underscore characters, and beginning with a non-digit). A conditional JML comment is guarded by one or more positive or negative keys (or both). The keys are placed just before the `@` character that is part of the opening sequence of the JML comment (the `//@` or the `/*@`). Each key is preceded by a '+' or a '-' sign, to indicate whether it is a positive or negative key, respectively. *No white-space is allowed*. If there is white-space anywhere between the initial `//` or `/*` and the first `@` character, the comment will appear to be a normal Java comment and will be silently ignored.

The keys are interpreted as follows. Each tool that processes the Java+JML input will have a means (e.g. by command-line options) to specify the set of keys that are enabled.

- If the JML annotation has no keys, the annotation is always processed.
- If there are only positive keys, the annotation is processed only if at least one of the keys is enabled.
- If there are only negative keys, the annotation is processed unless one of the keys is enabled.

- If there are both positive and negative keys, the annotation is processed only if (a) at least one of the positive keys is enabled AND (b) none of the negative keys are enabled.

JML previously defined one conditional annotation: those that began with `/*+@` or `//+@`. ESC/Java2 also defined `/*-@` and `//-@`. Both of these are now deprecated. OpenJML does have an option to enable the +-style comments.

The particular keys do not have any defined meaning in the JML reference manual. OpenJML implicitly enables the following keys:

- **ESC** : the `ESC` key is enabled when OpenJML is performing ESC static checking;
- **RAC** : the `RAC` key is enabled when OpenJML is performing Runtime-Assertion-Checking.
- **DEBUG** : The `DEBUG` key is not implicitly enabled. However it is defined as the key that enables the **debug** JML statement. That is the **debug** statement is ignored by default and is used by OpenJML if the user enables the DEBUG key.
- **OPENJML** : The OPENJML key is enabled whenever OpenJML is processing annotations (and presumably is not enabled by other tools).
- **KEY**: The `KEY` key is reserved for annotations recognized by the KeY tool [**?**]. It is ignored by OpenJML.

Thus, for example, one can turn off a non-executable assert statement for RAC-processing but retain it for ESC and for type-checking by writing //-RAC@ assert ...

F.3.3 Finding specification files and the refine statement

Discuss obsolete syntax somewhere - including the refines statement

JML allows specifications to be placed directly in the .java files that contain the implementation of methods and classes. Indeed, specifications such as assert statements or loop invariants are necessarily placed directly in a method body. Other specifications, such as class invariants and method pre- and post-conditions, may be placed in auxiliary files. For classes which are only present as .class files and not as .java files, the auxiliary file is a necessity.

Current JML allows one such auxiliary file per `.java` file or corresponding `.class` file. It is similar to the corresponding `.java` file except that

- it has a `.jml` suffix
- it contains no method bodies (method declarations are terminated with semicolons, as if they were abstract)
- TBD - field initializations?

The `.jml` file must be in the same package as the corresponding `.java` file and has the same name, except for the suffix. It need not be in the same folder, though the tail of the path to the folder containing the `.jml` file must still correspond to the package

containing the `.java` and `.jml` files. If there is no source file, then there is a .jml file for each compilation unit that has a specification. All the nested, inner, or top-level classes that are defined in one Java compilation unit will have their specifications in one corresponding `.jml` file.

The search for specification files is analogous to the way in which `.class` files are found on the *classpath*, except that the *specspath* is used instead. To find the specifications for a public top-level class *T*:

- look in each element of the *specspath* (cf. section TBD), in order, for a fully-qualified file whose name is *T*`.jml`. If found, the contents of that file are used as the specifications of *T*.
- if no such `.jml` file is found, look in each element of the *specspath*, in order, for a fully-qualified file whose name is *T*`.java`.

There are two (silent) consequences of this search algorithm that can be confusing:

- If both a `.jml` and a `.java` file exist on the specspath and both contain JML specification text, the specifications in the `.java` file will be (silently) ignored.
- If a `.java` file is listed on the command-line it will be compiled (for its Java content), but if it is not a member of an element of the *specspath*, it will (silently) not be used as the source of specifications for itself.

Obsolete syntax. The `refine` and `refines` statements are no longer recognized. The previous (complicated) method of finding specification files and merging the specifications from multiple files is also no longer implemented. The only specification file suffix allowed is `.jml`; the others — `.spec`, `.refines-java`, `.refines-spec`, `.refines-jml` — are no longer implemented.

In addition, the `.jml` file is now sought before seeking the `.java` file; if a `.jml` file is found anywhere in the specs path, then any specifications in the `.java` file are ignored. This is a different search algorithm than was previously used.

F.3.4 Model import statements

This section will be added later. Java `import` statements introduce class names into the namespace of a .java file. JML has a `model import` statement:

```
//@ model import ...
```

The effect of a JML model import statement is the same as a Java import statement, except that the names imported by the JML statement are only visible within JML annotations. If the model import statement is within a .jml file, the imported names are visible only within annotations in the .jml file, and not outside JML annotations and not in the .java file.

JML Keyword	Java annotation	class	interface	method	field declaration	variable declaration
code	Code			X		
code_bigint_math	CodeBigintMath					
code_java_math	CodeJavaMath					
code_safe_math	CodeSafeMath					
extract	Extract					
ghost	Ghost				X	X
helper	Helper			X		
instance	Instance					
model	Model					
monitored	Monitored					
non_null	NonNull			X	X	X
non_null_by_default	NonNullByDefault	X	X	X		
nullable	Nullable			X	X	X
nullable_by_default	NullableByDefault	X	X	X		
peer	Peer					
pure	Pure	X	X	X		
query	Query					
readonly	Readonly					
rep	Rep					
secret	Secret					
spec_bigint_math	SpecBigintMath					
spec_java_math	SpecJavaMath					
spec_protected	SpecProtected					
spec_public	SpecPublic					
spec_safe_math	SpecSafeMath					
static	Static					
uninitialized	Uninitialized					

Table F.3: Summary of JML modifiers. All Java annotations are in the `org.jmlspecs.annotation` package.

Note: Most tools only approximately implement this feature. For example, see FIXME for a discussion of this feature in OpenJML.

F.3.5 Modifiers

Modifiers are JML keywords that specify JML characteristics of methods, classes, fields, or variables. They are syntactically placed just like Java modifiers, such as `public`.

CHeck the table; add section references; add where allowed; indicate which are type modifiers; turn headings 90 degrees.

Obsolete syntax. JML no longer defines the modifier `weakly`.

F.3.6 JML expressions

Expressions in JML annotations are Java expressions with three adjustments:

- Expressions with side-effects are not allowed. Specifically, JML excludes
  - the ++ and – pre- and post- increment and decrement operations
  - the assignment operator
  - assignment operators that combine an operation with assignment (e.g., +=)
  - method invocations that are not explicitly declared `pure` (cf. §TBD)
- JML adds additional operators to the Java set of operators, discussed in subsection §**??** below.
- JML adds specific keywords that are used as constants or function-like expressions within JML expressions, discussed in subsection §**??** below

F.3.7 JML types

Specifications are sometimes best written using infinite-precision mathematical types, rather than the fixed bit-width types of Java. JML's arithmetic modes (§**??**) allow choosing among various numerical precisions. In this section we simply note the type names that JML defines.

All of the Java type names are legal and useful in JML: `int short long byte char boolean double real` and class and interface types. In addition, JML defines the following:

- `\bigint` - the type of infinite-precision integers, represented as java.lang.BigInteger during run-time checking
- `\real` - the type of mathematical real numbers, represented as TBD during runtime-checking
- `\TYPE` - the type of JML type objects

The familiar operators are defined on values of the `\bigint` and `\real` types: unary and binary + and -, *, /, %. Also, these types can be used in quantified expressions and variables of these types can be declared as ghost or model variables.

The set of `\TYPE` values includes non-generic types such has `\type(org.lang.Object)`, fully parameterized generic types, such as `\type(org.utils.List<Integer>)`, and primitive types, such as `\type(int)`. The subtype operator (`<:`) is defined on values of type `\TYPE`.

TBD - what about other constructors or acccessors of TYPE values

Appendix G

Statement translations

TODO: Need to insert both RAC and ESC in all of the following.

G.1 While loop

Java and JML statement:

```
//@ invariant invariant_condition ;
//@ decreases counter ;
while (condition) {
    body
}
```

Translation: *TODO: Needs variant condition, havoc information*

```
{
    //@ assert jmltranslate(invariant_condition) ;
    //@ assert jmltranslate(variant_condition) > 0 ;
    while (true) {
        stats(tmp,condition)
        if (!tmp) {
            //@ assume !tmp;
            break;
        }
        //@ assume tmp;
        stats(body)
```

```
    }
}
```

Appendix H

Java expression translations

H.1 Implicit or explicit arithmetic conversions

TODO

H.2 Arithmetic expressions

TODO: need arithmetic range assertions

In these, *T* is the type of the result of the operation. The two operands in binary operations are already assumed to have been converted to a common type according to Java's rules.

stats(tmp, **-** **a** *)* ==>
    stats(tmpa, **a** *)*
    T tmp = **-** *tmpa* **;**

stats(tmp, **a** **+** **b** *)* ==>
    stats(tmpa, **a** *)*
    stats(tmpb, **b** *)*
    T tmp = *tmpa* + *tmpb* **;**

stats(tmp, **a** **-** **b** *)* ==>
    stats(tmpa, **a** *)*
    stats(tmpb, **b** *)*
    T tmp = *tmpa* - *tmpb* **;**

stats(tmp, **a** `*` **b** *) ==>*
stats(tmpa, **a** *)*
stats(tmpb, **b** *)*
T tmp = tmpa `*` *tmpb* ;

stats(tmp, **a** `/` **b** *) ==>*
stats(tmpa, **a** *)*
stats(tmpb, **b** *)*
`//@ assert` *tmpb* `!= 0;` *// No division by zero*
T tmp = tmpa `/` *tmpb* ;

stats(tmp, **a** `%` **b** *) ==>*
stats(tmpa, **a** *)*
stats(tmpb, **b** *)*
`//@ assert` *tmpb* `!= 0;` *// No division by zero*
T tmp = tmpa `%` *tmpb* ;

H.3 Bit-shift expressions

TODO

H.4 Relational expressions

No assertions are generated for the relational operations `< > <= >= == !=`. The operands are presumed to have been converted to a common type according to Java's rules.

stats(tmp, **a** *op* **b** *) ==>*
stats(tmpa, **a** *)*
stats(tmpb, **b** *)*
T tmp = tmpa op tmpb ;

H.5 Logical expressions

stats(tmp, `!` **a** *) ==>*
stats(tmpa, **a** *)*

T tmp = ! *tmpa* ;

The && and || operations are short-circuit operations in which the second operand is conditionally evaluated. Here & and | are the (FOL) boolean non-short-circuit conjunction and disjunction.

stats(tmp, **a** && **b** *)* ==>

```
    boolean  tmp ;
    stats(tmpa, a )
    if ( tmpa ) {
        //@ assume tmpa ;
        stats(tmpb, b )
        tmp = tmpa & tmpb ;
    } else {
        //@ assume ! tmpa ;
        tmp = tmpa ;
    }
```

stats(tmp, **a** || **b** *)* ==>

```
    boolean  tmp ;
    stats(tmpa, a )
    if ( ! tmpa ) {
        //@ assume ! tmpa ;
        stats(tmpb, b )
        tmp = tmpa | tmpb ;
    } else {
        //@ assume tmpa ;
        tmp = tmpa ;
    }
```

Bibliography

[1] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML reference manual. Available from `http://www.jmlspecs.org`, September 2009.

Index

spec_protected, 4
spec_public, 4

old, 7
m CodeBigintMath, 75
m CodeJavaMath, 75
m CodeSafeMath, 75
m SpecBigintMath, 75
m SpecJavaMath, 75
m SpecSafeMath, 75
(*...*), 67
.jml files, 12
<:, 61
JML.informal(), 67
JML.lbl(), 64
JML.lblneg(), 64
JML.lblpos(), 64
<#=, 62
<#, 62
accessible clause, 43
assert statement, 50
assignable clause, 42
assume statement, 51
breaks clause, 45
callable clause, 44
captures clause, 44
choose_if clause, 44
choose clause, 44
continues clause, 44
diverges clause, 43
duration clause, 43
ensures clause, 42
extract clause, 44
forall clause, 43
ghost, 49
hence_by statement, 55
instance, 49
in, 49
maps, 49
measured_by clause, 43
model_program clause, 44
model, 49
monitored, 49
old clause, 43
or clause, 44
peer, 49
query, 49
reachable statement, 52
readonly, 49
rep, 49
requires clause, 42
returns clause, 44
secret, 49
signals_only clause, 42
signals clause, 42
uninitialized, 49
unreachable statement, 51
when clause, 43
working_space clause, 43
@Ghost, 49
@Instance, 49
@Model, 49
@Monitored, 49
@Peer, 49
@Query, 49
@Readonly, 49
@Rep, 49
@Secret, 49
@Uninitialized, 49
\TYPE, 29
\bigint, 28
\duration, 72
\elemtype, 69
\exception, 63

\fresh, 66
\invariant_for, 70
\is_initialized, 69
\key, 64
\lblneg, 64
\lblpos, 64
\lbl, 64
\lockset, 71
\max, 71
\nonnullelements, 65
\not_modified, 71
\old, 63
\past, 63
\pre, 63
\reach, 71
\real, 29
\result, 62
\space, 73
\typeof, 68
\type, 68
\working_space, 72
\bigint, 74
\real, 74

abstract data type, 4, 6
abstract fields, 4
abstract value, 6
abstract value, of an ADT, 4
ADT, 4
Arithmetic modes, 74
axiom, 36

behavior, 3
behavioral interface specification, 3
Borgida, 3

Cheon, 4
code_bigint_math, 46, 75
code_java_math, 46, 75
code_safe_math, 46, 75
constraint clause, 35

data groups, 47
datatype, 6
debug statement, 53

Eiffel, 3
extract, 46

field specifications, 47
Fitzgerald, 6
frame axiom, 3
Fresco, 4
function, 45

ghost fields, 35
Guttag, 3, 6

Handbook, for LSL, 6
Hayes, 4, 6
helper, 45
Hoare, 6
Horning, 3, 6

informal expression, 67
initializer, 35
initially clause, 35
interface, 3
interface specification, 3
interface, field, 3
interface, method, 3
interface, type, 3
invariant clause, 34
ISO, 6

java.math.BigInteger, 74
Jones, 6

Lamport, 3
Larch, 3
Larch Shared Language (LSL), 3
Larch style specification language, 3
Larch/C++, 6
Larsen, 6
Leavens, 3
Leino, 3
Liskov, 6
loop specifications, 53
LSL, 3
LSL Handbook, 6

method, behavior of, 3
Meyer, 3, 6
model, 45

model classes, 35
model fields, 35
model import statement, 30
model methods, 35
model-oriented specification, 3
modifiers, 16, 91
monitors_for clause, 36

Nelson, 3
non_null, 45, 47
non_null_by_default, 34
nullable, 45, 47
nullable_by_default, 34

operation, 6
operator, of LSL, 6
options, 46

package-info.java, 30
Parnas, 6
peer, 48
postcondition, 3, 6
postcondition, exceptional, 3
postcondition, normal, 3
precondition, 3, 6
pure, 34, 45

query, 45

readable if clause, 36
represents clause, 35
Rosenblum, 3

Saxe, 3
secret, 45
set statement, 53
skip_esc, 46
skip_rac, 46
SkipRac, 46
spec_bigint_math, 75
spec_java_math, 75
spec_protected, 45, 47
spec_public, 45, 47
spec_safe_math, 75
Specification inheritance, 15
specification of fields, 47
specification, of interface behavior, 3
Spivey, 4, 6
statement specification, 54
static_initializer, 35

trait, 6
trait function, 6
type, abstract, 6

value, abstract, 6
VDM, 6
VDM-SL, 6
vocabulary, 3

Wills, 4
Wing, 3
writable if clause, 36

Z, 4, 6