Does your software do what it should?

User guide to specification and verification with the Java Modeling Language and OpenJML

David R. Cok
david.r.cok@gmail.com

DRAFT April 8, 2022

This document is being actively expanded, edited and reviewed. Comments are welcome. We intend for a final version to be available by fall 2022.

Foreword

Gary write this?

Preface

The Java Modeling Language (JML) project started in about 1997 with the goal of enhancing the capability of specification and automated verification to improve the development of software. A current review article [19] summarizes some of the experience and challenges of this project.

The OpenJML tool, in development since 2006, performs the work of checking that specifications written in JML match implementations written in Java. The incarnation of that tool described in this document is based on OpenJDK, is compatible with Java 17, and has been used in both industrial and academic applications. The JML language and the OpenJML tool are similar in concept to the specification languages and tools for other programming languages; they thus fit within the wider research and development endeavor to create specification and verification capabilities that work well with the day-to-day work of conventional software programming.

This book itself is just the user guide and reference manual for OpenJML. The most current version of this document is maintained on-line at www.openjml.org/documentation/OpenJMLUserGuide.pdf.

- It is not a language guide. For that see the JML Reference Manual: https://www.openjml.org/documentation/JML_Reference_Manual.pdf.
- It is not a tutorial. For that see the online OpenJML tutorial at https://www.openjml.org/tutorial.
- It is not a discussion of how to develop the source code for the tool. For that see the github project at https://github.com/OpenJML/OpenJML.
- It is not general guide to research and projects related to JML. For that see the JML project website at http://www.jmlspecs.org.

- It is not a comparison to other tools. One other relevant project is the KeY project: https://www.key-project.org/ — including a book about KeY: https://www.key-project.org/thebook2/

OpenJML, though developed primarily by David R. Cok, has benefited from many sources:

- The JML initiative, started and overseen by Gary Leavens.
- A long history of research on the Java Modeling Language itself, as reflected in the publications list on the JML project web site: http://www.jmlspecs.org.
- The work on previous and succeeding languages and tools for other programming languages, most notably
  - the Frama-C project (https://www.frama-c.com)
  - and Dafny (https://github.com/dafny-lang/dafny).
- Previous work on JML tools preceding OpenJML, such as EscJava [12], EscJava2 [8], and the ISU suite of tools [6].
- Occasional individual contributors to OpenJML itself.
- The OpenJDK compiler framework on which OpenJML is built: https://www.openjdk.org.
- The cross-fertilization with colleagues at the KeY project: https://www.key-project.org/.

Contents

Chapter 1

Introduction to JML and OpenJML

The Java Modeling Language [18] has been evolving since the beginning of the project in 1997. The project as a whole includes the specification language definition, research on language features for specification, development of tools (such as OpenJML), application of JML and OpenJML to academic and industrial problems, and encouraging their use in education.

JML is widely known and is the inspiration for analogous tools for languages other than Java, such as ACSL [1] for C, ACSL++ [1] for C++, Spec# [4] for C#, SPARK [2] for Ada, Stainless/Leon [23, 5] for Scala, and Dafny[21]. JML has evolved considerably over the years, as Java has evolved. The JML Reference Manual (2nd edition) [9] is a substantial rewrite of the original Draft Reference Manual [20] in order to include many new features (corresponding to Java language features) and new developments in program specification and verification.

Similarly, tools to support JML have evolved. The first tools relied on infrastructure that proved unmaintainable over time, as Java changed. Consequently, when OpenJDK became available in 2006, the JML project adopted OpenJDK as the compiler framework on which to build OpenJML. The first series of versions of OpenJML supported Java 8. In 2020, work was started to upgrade to Java 16ff. This endeavor required substantial internal reorganization because of the introduction of modules as a Java language feature and the use of modules in

the OpenJDK source code itself. The current version of OpenJML is easier to install and run than previous versions. The source code, releases and development materials of OpenJML are hosted on GitHub, at https://github.com/OpenJML The project as a whole is open source, with the OpenJML tool, like OpenJDK, publicly available under the GPLv2 license.

There are three companion resources that you should be aware of in using JML and OpenJML:

- **The Java Modeling Language (JML)** is a specification language for Java programs. There is a reference manual for JML on-line at https://www.openjml.org/documentation/JML_Reference_Manual.pdf.
- OpenJML is a tool for checking Java program implementations against their JML specifications. This document, the user guide (reference manual) for OpenJML, describes how to use the tool: installation, execution, command-line options and the like. The most current version of this document is on-line at https://www.openjml.org/documentation/OpenJMLUserGuide.pdf.
- A **tutorial** with lessons on using JML and OpenJML is on-line at https:/openjml.org/tutorial.

Additional resources are listed in §1.3.

The most significant, well-supported other tool for JML is the KeY tool — http://www.key-project.org/

1.1 Why specify? Why check?

Software is hard to write correctly. In some applications, software can be safety-, security- or life-critical. Many tools and processes have been promoted and tried to create better software: testing frameworks, coverage measures, requirements processes, careful development processes, agile development processes, fuzzing, separate testing teams, and so on. Deductive verification (also known as formal methods) is another such technique. It has the advantage of applying automated logic provers to check the consistency of machine-readable specifications and software implementations. It has the disadvantage of requiring the work of writing specifications in logical form along with the actual software implementation. Even just the added rigor and careful design work needed to write a

verifiable specification can improve the quality and correctness of the resulting software artifacts. And as any compiler reminds an engineer, tools that check our work invariably find errors to correct; the same is true for static specification checking tools.

Deductive verification is a form of *static analysis* in that it checks software without running it. However, most tools labeled as static analysis tools check things like code style or identify bug patterns or bad-smelling code. Deductive verification takes this much further by logically reasoning about what the code actually does, to find input sets that lead to crashes or to violations of expected behavior.

Although (static) deductive verification is more rigorous than (dynamic) testing because verification uses automated logic tools and can validate all execution paths (not just those for which there are test cases), it is not perfect: in the end, the implementation and the specification both must reflect what the software writer intended, and that requires careful manual review along with automated tooling.

This document describes a tool, OpenJML, that performs deductive verification: it checks that software written in Java is consistent with specifications written in the Java Modeling Language (JML) [9, 18]. There are other tools that perform the same task for other programming languages, such as ACSL for C [1], ACSL++ for C++ [1], Spec# [4] for C#, SPARK [2] for Ada, Leon/Stainless [23] for Scala, Dafny [21] (for Dafny). There is also the KeY tool [17] for Java.

1.2 Background on OpenJML

OpenJML is a tool for checking that the source code of a Java program is consistent with specifications for that code written in the Java Modeling Language (JML). The tool parses and type-checks the specifications and performs static or run-time checking of the implementation code and the specifications. Because OpenJML is built on the Java compiler, it is also able to do pure Java compilation, which it uses to compile Java programs with extra runtime checks.

OpenJML, like verification tools for other languages, checks that the code that implements a programming language method is consistent with the specifications for that method. To do this, OpenJML converts both the method implementation and the method specifications, along with the specifications of called

methods, into a logical form. A separate tool, an SMT proof tool, is then automatically invoked to see if there is any possible execution of the implementation that would violate the specification. If there is, a counterexample to correct functioning is reported to the tool user; if not, that method is considered verified. If the source code + specifications for all the methods in the program are equivalently verified (and verified to terminate), then the program as a whole can be soundly considered to obey its specifications.

Tools like OpenJML can only check that the code and specifications are *consistent*, that is, that the code behaves as the specifications state; it is possible that the code and specifications, although consistent with each other, together are incorrect when compared to the behavior that the software engineer actually desires. Thus manual review that the formally stated specifications are complete and match informal or natural language specifications is also necessary. But even if the functional specifications are not complete, OpenJML, and tools like it, can assure that no runtime exceptions will be generated by any permitted execution of the program.

This list shows the functionality present or anticipated in OpenJML:

- parse and typecheck all of Java: Java is parsed through Java 17, as implemented in OpenJDK
- parse JML specifications for Java programs: all of JML is parsed, as defined by the JML Reference Manual v2
- typecheck all of JML, as described in this document and the JML Reference Manual
- static checking that Java code is consistent with the JML specifications: implemented
- runtime checking of JML specifications: still in progress for Java 17
- interacting with OpenJML programmatically from a host program: anticipated
- JML specifications included in javadoc documentation: planned
- JML specification inference: partially present with more in progress
- automatic test generation, based on JML specifications: planned
- symbolic execution of Java + JML programs: perhaps

Current OpenJML is a command-line tool available on MacOS, Linux, and on Windows under WSL.

OpenJML was constructed by extending OpenJDK, the open source Java compiler, to parse and include JML constructs in the abstract syntax trees representing the Java program. Using OpenJDK was a design decision made when OpenJDK became available. Precursor tools were built on other frameworks: EscJava2 on a custom-built Java compiler; ISU tools on MultiJava. But both of these required far too much developer effort just to keep up with changes in Java. Other frameworks were considered, such as the Eclipse compiler. The choice of OpenJDK has been validated by the strong and continuing support for OpenJDK as Java has evolved.

1.3 Other resources

There are several useful resources related to JML and OpenJML:

- http://www.openjml.org contains a set of on-line resources for OpenJML, including the tutorial at http://www.openjml.org/tutorial
- The source code, releases, and issue list for OpenJML are maintained in the GitHub project at http://www.github.com/OpenJML. This project also contains related material such as the test suite, Java library specifications, SMT solvers
- The OpenJML GitHub project wiki contains information relevant to *developing* OpenJML: https://github.com/OpenJML/OpenJML/wiki .
- http://www.jmlspecs.org is a web site containing information about JML, including references to many publications, other tools, and links to various groups using JML.
- https://www.openjml.org/documentation/OpenJMLUserGuide.pdf is the most current version of this document
- https://www.openjml.org/documentation/JML_Reference_Manual.pdf is the most current version of the JML reference manual
- http://www.jmlspecs.org/OldReleases/jmlrefman.pdf is the first version reference manual for JML [20], which is being superseded by the document mentioned in the previous bullet

- the original JML tools and some other older (typically obsolete and no longer maintained) JML projects are contained in the jmlspecs sourceforge project at http://sourceforge.net/projects/jmlspecs.

There are also other tools that make use of JML. An incomplete list follows:

- The KeY tool — http://www.key-project.org/
- The previous generation of JML tools prior to OpenJML is available at http://www.jmlspecs.org/download.shtml.
- Other tools and projects listed at jmlspecs.org.
- A previous sourceforge project for OpenJML has been discontinued in favor of the GitHub project.

Various mailing lists and discussion groups answer questions and debate JML language syntax and semantics.

- The issues list at https://github.com/JavaModelingLanguage/RefMan/issues is the place for discussions of JML syntax and semantics, including questions about JML.
- The issues list at https://github.com/OpenJML/OpenJML/issues is the place for discussion and questions about (and problems with) OpenJML.

1.4 Sources of Technology

The design and implementation of OpenJML uses and extends many ideas present in prior tools, such as ESC/Java[10] and ESC/Java2[8], and from discussions with builders of tools such as Spec#[4], Boogie[3], Dafny[21], Frama-C[13], KeY[17], ACSL[1], and the Checker framework[11]. It also benefits from many advances in specification technology over the past couple of decades.

1.5 License

The OpenJML command-line tool is built from OpenJDK, which is licensed under GPLv.2 (http://openjdk.java.net/legal/). Hence OpenJML is correspondingly licensed as GPLv.2.

The source code for OpenJML and any corresponding modifications made to OpenJDK are available from a GitHub project: https://github.com/OpenJML.

1.6 Use of this document

This document is meant as a resource, in the spirit of most reference manuals, rather than a text to be read straight through. The best approach is to work through the on-line tutorial, with the JML and OpenJML reference manuals at hand to provide detail when you need it. Once you understand the introductory concepts, then more thorough reading of the reference manuals will alert you to advanced features that you may need. The JML Reference Manual is the guide to the definition of JML features. This document provides information on how to use OpenJML to do static and runtime verification.

Downloading code samples

Chapter 2

Installation

2.1 Installing OpenJML

The OpenJML releases are kept in the OpenJML GitHub project; the installation file is a simple .zip file. There are different builds for different platforms. Currently, MacOS, Linux (Ubuntu), and Windows on Cygwin are supported.

- Find the latest release of the highest number series, currently 17, at https://github.com/openjml/openjml/releases .
- Download the artifact for your platform. It is a .zip file.
- Create a clean folder of your choice and unzip the downloaded release into it. The installation folder, call it *OJ*, will contain files and folders such as `openjml`, `tutorial`, etc.
- The executable (a bash script) to run is *OJ*`/openjml`. Do not move this file out of its location within the installation, as it uses its location to find resources needed by OpenJML. You can write a script to delegate to *OJ*`/openjml`, storing your script in some place on your PATH, if you like. Or you can put *OJ* on your PATH. If you use a symbolic link to point to the *OJ*`/openjml` executable, then you need the utility `realpath` in your environment; on MacOS you may need to install that explicitly, for example using `brew install coreutils`.

The installation includes some demo and tutorial files, in the *OJ*`/demo` and

OJ/tutorial folders. The tutorial files are meant to be used with the on-line tutorial at https://www.openjml.org/tutorial.

You can give OpenJML a quick trial by running the command

```
OJ/openjml --esc OJ/tutorial/T_ensures2.java
```

This command should give some error messages identifying some specification errors in the T_ensures2.java file.

2.2 Organization of the installation

Check that this is accurate

The installation contains the following, all within the installation folder (*OJ*):

- The executable openjml, which executes the OpenJML tool itself. It is a replacement for javac.
- The executable openjml-java, which is a replacement for java: it executes compiled Java programs with runtime-assertion-checks and includes the runtime libraries necessary to do so.
- The library jmlruntime.jar, which must be included with a RAC-compiled program when run with a conventional java.
- The executable mac-setup, which turns off MacOS warnings about unknown executables, if necessary
- The folder tutorial, which contains the files used in the JML/OpenJML tutorial (https://www.openjml.org/tutorial).
- The folder demos, which contains other demo files.
- Copies of this *OpenJML Users' Guide* and the *JML Reference Manual v2* current at the time of the build release.

2.3 Local customization

OpenJML can be customized to your local environment as described in §4.3. Local properties are specified in a openjml.properties file, stored in the same directory as openjml or in the user's home directory.

The `openjml.properties` file can be used to indicate default command-line arguments and other local properties used by the tool. The installation includes the file `openjml.properties-template`, which can be copied and customized to create `openjml.properties`.

SMT solvers are needed if you intend to use the static checking capability of OpenJML (cf. §7). Recommended solvers are included in the installation package and are used by default. If you wish to use an alternate SMT solver, the location of the solver can be specified on the command-line or, more easily, in the `openjml.properties` file. For example, if the Z3 4.3 solver is located in your system at absolute location *<path>*, then include the following line in the `openjml.properties` file:

`org.openjml.prover.z3_4_3=`*<path>*

The details of the `openjml.properties` file are described in §4.3.

Chapter 3

The OpenJML Command-line Tool

3.1 Command-line structure

OpenJML is a conventional command-line tool. In fact it acts much like the Java compiler (`javac`), but with additional command-line options and capabilities: the command-line consists of space-separated arguments, each of which is a file-system path or an option or an option followed by the option's value.

The options are all listed in Tables 5.1 and 5.2; the tables have links to where the options are described in relevant sections throughout this document. The general form of options and their values is described in §5.1. In short

- options begin with one or two hyphens
- options may be boolean-valued or have a (string) value
- later options in the command-line override earlier ones of the same name
- options and file paths may be freely intermixed on the command-line

3.2 Files and Folders

Besides options, the Java compiler only allows files to be listed on the command-line. OpenJML allows listing folders as well, using the **-dir** and **-dirs** options (cf.

§5.4). A folder on the command-line is replaced by all the `.jml` files within that folder and its subfolders, recursively.

As described later in §4.1, JML specifications for Java programs can be placed either in the `.java` files themselves or in auxiliary `.jml` files. The format of `.jml` files is defined by JML. OpenJML type-checks `.jml` files along with the corresponding `.java` files or `.class` files, as described in §4.2).

3.3 Output

OpenJML sends all of its output to Java's `System.out`. That output consists of error messages, verification failure messages, warnings, and informational output, such as progress indications. No output generally means success, though it can mean a lengthy operation.

In addition, when operating as a compiler (e.g., for runtime assertion checking), OpenJML produces class files in the same manner as `javac` would.

3.4 Exit values

A command-line tool running in a shell interpreter is expected to emit an integer exit code on completion, indicating success or various kinds of failure. OpenJML emits one of these values on exit:

- 0 (`EXIT_OK`) : successful operation, no errors, there may be warnings
- 1 (`EXIT_ERROR`) : normal operation, but with parsing or type-checking errors
- 2 (`EXIT_CMDERR`) : an error in the formulation of the command-line, such as invalid options
- 3 (`EXIT_SYSERR`) : a system error, such as out of memory
- 4 (`EXIT_ABNORMAL`) : a fatal error, such as a program crash or internal inconsistency, caused by an internal bug
- 5 (`EXIT_CANCELLED`) : indicates exit because of user initiated cancellation
- 6 (`EXIT_VERIFY`) : indicates exit because of verification failures

The JML option –**verify-exit** allows the user to set an alternate value for the exit code in the case of verification failures, such as 1 to count them the same as errors, or 0 to count them the same as warnings).

The Java option **-Werror** indicates to treat all warnings as errors. The Java option **-nowarn** suppresses warnings.

To elaborate the alternatives:

	exit code without **-Werror**	exit code with **-Werror**
Java or JML errors	1, 2, 3, or 4	1, 2, 3, or 4
Java or JML warnings only	0	1
verification failures, with or w/o warnings no **–verify-exit**	6	6
verification failures and no warnings and **–verify-exit** not 0	value of **–verify-exit**	value of **–verify-exit**
verification failures and with warnings or **–verify-exit=0**	value of **–verify-exit**	1

Table 3.1: Exit codes

Chapter 4

OpenJML Concepts

4.1 Specifications in .java and .jml files

JML allows specifications for Java methods and classes to be placed either directly in the .java source file or in an auxiliary .jml file. The latter is required if there is no source file, such as for a library, or if the source file may not be modified, such as for a highly controlled project.

The format of a .jml file is very much like the corresponding .java file, with the largest difference being that the Java implementations of methods are omitted in the .jml file. Other differences are described in the JML Reference Manual.

If .jml files are used, the question then is where are they located and how does a tool find them. That process is described in the following section (§4.2).

4.2 Finding files and classes: class, source, and specs paths

A key concept to understand is how class files, source files, and specification files are found and used by the OpenJML tool. Java uses a *classpath* and a *sourcepath* to locate compiled and source files; these are designated by the **-classpath** (or **-cp** or **--class-path**) and **-sourcepath** (or **--source-path**) (Java) options. OpenJML adds a *specspath* to find specification files, which is designated by the **--specs-path** OpenJML option.

The files and folders listed on the command-line must be given as absolute paths or paths relative to the current working directory. But these files may (most assuredly will) contain references to other classses. The *classpath* and *sourcepath* are used to resolve these references to classes as compiled `.class` or source `.java` files..

Each of these paths is a sequence of file system paths identifying folders or jar files. When Java tools are looking for compiled class files it will look in each of these folders on the *classpath* in turn; similarly source code files are looked for in the *sourcepath*. If a Java class has both a compiled and source version available, the -**Xprefer** option determines which is used.

Recall that the folders on the class and source paths represent the root of the package for that class. That is, a class `p.AA` (in package `p`) must have a class file at `X/p/AA.class` with X on the classpath or a source file `Y/p/AA.java` with Y on the sourcepath. Specification files are named and stored in the file system in the same way. The classpath may also contain jar files that contain the files being sought.

The OpenJML tool also needs to find specification files. These can be either `.java` or `.jml` files; if it is a `.jml` file, it will have the same file name (with a `.jml` extension) and package as the Java class. Whenever a class, either source or compiled, is read into OpenJML, the tool will look for a corresponding specification file on the *specspath*, which is set by the --**specs-path** option. First, the full specspath is searched for the corresponding `.jml` file; if it is not found, then the specspath is searched again for a corresponding `.java` file. If still not found and the class was read from a source file on the command-line, then a `.jml` file is looked for in the same folder as the `.java` file; if that is not found then the `.java` file from the command-line is used. If no source or specification file is found (that is, there is only a `.class` file, then a default set of specifications is used, as defined by JML.

Most often, the user need not set all of these paths because there are convenient defaults:

- `classpath`: The OpenJML classpath is set using one of these alternatives, in priority order, with the system library always being added as well:
  - the argument to the OpenJML command-line option -**classpath**
  - the value of the Java property `org.jmlspecs.openjml.classpath`
  - the value of the system environment variable `CLASSPATH`

  - the default, which is the current working directory (plus the system library)
- `sourcepath`: The OpenJML sourcepath is set using one of these alternatives, in priority order:
  - the argument of the OpenJML command-line option -**sourcepath**
  - the value of the Java property `org.jmlspecs.openjml.sourcepath`
  - the value of the OpenJML classpath (as determined above), without the system libraries (which are all `.class` files)
- `specspath`: The OpenJML specifications path is set using one of these alternatives, in priority order, with the locations of the system library specifications always appended:
  - the argument of the OpenJML command-line option --**specs-path**
  - the value of the Java property `org.jmlspecs.openjml.specspath`
  - the value of the OpenJML sourcepath (as determined above)

Note that with no command-line options or Java properties set, the result is simply that the system CLASSPATH (and absent that, the current working directory) is used for all of these paths. A common practice is to simply use a single directory path, specified using the system CLASSPATH or on the command-line using -**cp**, for all three paths.

Despite any settings of these paths, the Java system libraries are always effectively included in the classpath; similarly, the JML library specifications that are part of the OpenJML installation are automatically appended to the specifications path. Placing an alternate set of specification files on the specspath effectively replaces any built-in system library specifications.

OpenJML will warn about folders on the specspath that do not exist. The warning can be suppressed with the option --**no-check-specs-path**.

A common working style has specifications written directly in `.java` files and not using separate `.jml` files. In this case the user should be sure that the specspath includes the sourcepath (which it does by default). Otherwise, OpenJML will not find the `.java` file when looking for specifications and will then use default specs, confusingly ignoring any specifications in the `.java` file.

There are a number of common scenarios:

- Java source file on the command-line with a corresponding JML file on the specifications path: the JML file is used as the specification of the Java class, *with any JML content in the Java source file completely ignored.*

- Java source file on the command-line with no corresponding JML file on the specifications path: the Java source file is used as its own JML specification; if it contains no JML content, then default specifications are used.
- Java class file on the classpath or in the Java system library (referred to by files on the command-line) and a corresponding JML file on the specifications path: the JML file is used as the specifications for the class file. Any corresponding source file on the sourcepath or command-line is ignored. Check that the source file is ignored even with -Xprefer
- Java class file on the classpath or in the Java system library (referred to by files on the command-line), no corresponding Java source file on the sourcepath or command-line, and no corresponding JML file on the specifications path: the class file is used with default specifications.

There are two complicated scenarios:

- a source file on the command-line is not on the sourcepath and there is an additional, different source file for the same class on the sourcepath
- two instances of a source file for the same class are on the sourcepath, with the one later in the sourcepath appearing on the command-line

In these two scenarios, one `.java` file is used as the source code and another as specification. If the two files define different methods or contain different specification text, OpenJML will likely issue error messages that may be confusing until the user figures out that there are two distinct files. This situation is likely an error and should be avoided.

4.3 OpenJML Options, Java properties and the `openjml.properties` file

The OpenJML tool is controlled by a variety of options, just as many other tools are. The general rules about options are presented in §5.1 and the implemented options are described in detail throughout this document; here we describe how the options can be set using properties rather than on the command-line.

OpenJML options interact with Java properties. Java properties can be used to set OpenJML options without needing to state them on the command-line each time. Java properties are typical key-value pairs of two strings. Values for boolean

options can be stated using the strings true and false. A typical use of properties in OpenJML is to record characteristics of the local environment that vary among different users or different installations. But they can also be used to set initial values of options, so they do not need to be set on the command-line.

OpenJML loads properties from specified files placed in several locations. It loads the properties it finds in each of these, in order, so later definitions supplant earlier ones.

- Properties defined by environment variables as described below
- A `openjml.properties` file in the OpenJML installation directory, if any
- The first `openjml.properties` file on the classpath, if any
- A `openjml.properties` file in the user's home directory (the value of the Java property `user.home`), if any
- A `openjml.properties` file in the current working directory (the value of the Java property `user.dir`), if any

Check the reading of openjml.properties.

Then the value of any property whose name has the form `org.openjml.`*option* is used to set the value of the *option* (leaving off the initial 1 or 2 hyphens). And then, finally, the options given on the command-line override any previously given values.

The format of a `.properties` file is defined by Java[1]. These are simplified statements of the rules:

- Lines that are all white space or whose first non-whitespace character is a `#` or `!` are comment lines
- Non-comment lines have the form *key*=*value* or *key*`:`*value*
- Whitespace is allowed before the key and between the key and the = or `:` character and between the = or `:` character and the value
- The value begins with the first non-whitespace character after the = or `:` character and ends with the line termination. This means that the value may include both embedded and trailing white space. (The presence of trailing white space in key-value pairs can be a difficult-to-spot bug.)

The properties that are currently recognized are these:

- `org.openjml.defaultProver` - the value is the name of the prover (cf. §4.4) to use by default

[1] https://docs.oracle.com/javase/8/docs/api/java/util/Properties.html#load(java.io.Reader)

- `org.openjml.prover.`*name*, where *name* is the name of a prover, and the value is the file system path to the executable to be invoked for that prover (cf. §4.4)
- `org.openjml.`*option*, where *option* is the name of an OpenJML option (without any leading hyphens)

The format of a shell environment variable is (unfortunately) slightly different, because such variables may not contain periods or hyphens. So to set an option named `--opt` to a value `val`, define the environment variable `OPENJML_opt=val`, where any hyphens in *opt* are replaced by underscores.

For example, if you are tired of always writing **--esc** when invoking `openjml`, you can change the default for the **--command** option, which is usually `check`, to `esc` by one of these:

- `OPENJML_command=esc openjml tutorial/T_ensures2.java`
  — temporary change just for this line
- `export OPENJML_command=esc; openjml tutorial/T_ensures2.java`
  — change applies to the remainder of the shell
- put `org.openjml.option.command=esc` in a `openjml.properties` file in your home directory (or the current working directory, or the installation directory) — change applies until the line is removed from the `openjml.properties` file.

The OpenJML distribution includes a file that contains stubs for all the recognized options: `openjml-template.properties` . You may copy that file, rename it as `openjml.properties`, and edit it to reflect your system and personal configuration, and put it in one of the designated locations. (If you are an OpenJML developer, take care not to commit your local `openjml.properties` file into the OpenJML shared GitHub repository.)

Does the template file really have all of these?

4.4 SMT provers

The static checking capability of OpenJML uses SMT solvers to discharge proof obligations stemming from the specifications and implementation of a program. The SMT solvers are not part of OpenJML itself. However, a selection of solvers is shipped with an OpenJML release and one of these is used by default.

If you want to use a different solver, you need to set these properties:

- `org.openjml.defaultProver` to give the name of a prover (e.g., `z3-4.3`)
- `org.openjml.prover.`*name*, where *name* is the name of a prover, and the value is the file system path to the executable to be invoked for that prover
  (e.g., `org.openjml.prover.z3-4.3=...`)

Different solvers have different properties. They support different SMT logics; for example, some do not support quantifiers, others may not support real arithmetic. They certainly also have different runtime and memory performance and different success rates at finding answers to proof obligations.

Currently, OpenJML **works best with Z3 v4.3.1, which is shipped with** OpenJML**, and is the default solver.**

4.5 Conditional JML annotations (-keys option)

JML defines a mechanism for controlling which JML annotations are used by tools (see the JML Reference Manual for more detail):

- Syntactically, a JML annotation comment can be enabled or disabled by positive or negative keys, as in `//+`*key*`@` and `//-`*key*`@`, where *key* is a Java identifier. See the JML Reference Manual for details.

This conditional annotation relies on the *key* being defined or not. OpenJML defines keys using the **-keys** option. The value of this option is a comma-separated list of identifiers, each of which is then a defined key. Like other options, a property (`org.openjml.option.keys`) can be defined to avoid adding options to the command-line.

In OpenJML,

- the key `OPENJML` is enabled by default in the OpenJML tool
- the keys `ESC` and `RAC` are enabled when the respective OpenJML tools are being executed
- the key `DEBUG` is reserved but is disabled by default

- the key KEY is reserved for the use of the KeY ([17]) tool and is disabled by default in OpenJML
- all other keys are disabled by default in OpenJML

Keys are case-sensitive. However reusing differently-cased versions of keys for different purposes is discouraged, including differently cased versions of the above. For example, the identifier KeY should be considered a reserved key along with KEY.

Two simple uses are these:

```
1 //+OPENJML@ requires x;
2 //-RAC@ ensures y;
```

The first line, with the + sign, is ignored in all situations except when OPENJML is defined as a key. The second line, with the – sign, is always enabled except when RAC is defined as a key. This second use case is quite commonly used to exclude from runtime-checking JML features that have a lengthy runtime or are non-executable.

4.6 Annotations and the runtime library

JML optionally uses Java annotations as introduced in Java 1.6 as an alternate way to specify modifiers. For example, a method can be declared pure either with the /*@ pure */ JML modifier or the @Pure Java annotation.[2] JML-defined annotation classes are in the package org.jmlspecs.annotation. In order for files using these annotations to be processed by Java, the annotation classes must be on the classpath (just like any other annotation classes). They may also be required when a compiled Java program that uses such annotations is executed. In addition, running a program that has JML runtime assertion checks compiled in will require the presence of runtime classes that define utility functions used by the assertion checking code.

Both the annotation classes and the runtime checking classes are provided in a library named jmlruntime.jar. The distribution of OpenJML contains this library.

[2]There are many annotations defined that are not used or not implemented. For example, a @Requires annotation was introduced as an experiment in writing preconditions with annotations, but not subsequently adopted into JML.

When the `openjml` and `openjml-java` executables are used to compile and run a Java program, both the annotations and the runtime utilities are automatically available, as they are built-in to those tools. It is possible to supplant the OpenJML-supplied versions of these classes by putting an alternative on the classpath.

If instead the conventional `java` tool is used to run a RAC-compiled executable, then the `jmlruntime.jar` library must be added to the classpath. An alternate library that provides at least the same capabilities may be used instead.

4.7 Defaults for binary classes

TODO: Say more

4.8 Redundancy in JML and OpenJML

JML has a few features that explicitly allow redundancy. Many keywords, such as `ensures`, have an alternate version, `ensures_redundantly`. The goal is to be able to state an equivalent assertion but in an alternate form that may be more understandable or provable. Similarly, the `implies_that` and `for_example` specification cases are not intended to state new behavior specifications, but rather to state implications or examples of behavior already given.

Although the semantics of these redundant specifications is that they be provable from other specifications, OpenJML currently

- treats the redundant keywords precisely like their non-redundant counterparts and
- ignores the `implies_that` and `for_example` specification cases.

4.9 Nullness and non-nullness of references

4.9.1 Background on non-null annotations and types

Whether or not references (or pointers) are null is a key source of programming faults in many programming languages. And an Optional type just hides the question in a different construct. So much so that newer languages (e.g., Dafny, Kotlin) are building in the concept of non-null types. Java itself has no

provision for non-null types, but various tools (e.g., Checker framework) have implemented Java annotations (`@NonNull`, `@Nullable`) to impose a statically-checkable, non-null type framework on top of Java.

JML has had from the beginning (before Java annotations were added to the Java language) the `non_null` and `nullable` modifiers, which indicated which variable, field, formal parameter, and method return values were or were not allowed to be null; tools supporting JML have always implemented verification checks of these restrictions.

A further development is the introduction of *type annotations* in Java. Now not only declarations, but all uses of a type name can be annotated — types used in declarations, in casts, in type parameters—anywhere a reference type name is permitted. Combined with the `@NonNull` and `@Nullable` annotations now being defined as type annotations, JML has a true non-null subtype for each reference type. There is one difficulty in that there are multiple packages that define these annotations: JML has them in `org.jmlspecs.annotation`, the Checker framework has them in `org.checkerframework.checker.nullness.qual`, the javax additions in `javax.annotation.Nonnull`. In fact, the Checker framework documents a long list of such annotations:
(https://checkerframework.org/manual/#nullness-related-work)

For now, OpenJML recognizes the JML annotations and has a future goal of recognizing the more important of other annotations.

The formal details of annotations, including type annotations, are in the Java Language Specification:

https://docs.oracle.com/javase/specs/jls/se17/html/jls-9.html#jls-9.7.4

A more understandable discussion is found in JSR-308:

https://checkerframework.org/jsr308/specification/java-annotation-design.html

4.9.2 JML features for nullness

JML adopted the semantics that *reference types are non-null by default*, even though that is not the Java default. Soundness? This policy helps identify null reference problems early and requires explicit specification that a reference may be null. Resolve the default for local variables.

Nullness defaults are determined as follows:

- A class may be marked with one of the modifiers

`non_null_by_default` or `nullable_by_default`

or the annotations

`@NonNullByDefault` or `@NullableByDefault`

to indicate whether type names in the class are non-null or nullable by default.

- A class not so marked takes its default from the modifiers or annotations on the innermost enclosing class that has such a modifier or annotation.
- In the absence of a marked enclosing class, the default is taken from the command-line options (or properties) — either the option **--nonnull-by-default** or **--nullable-by-default**
- And in the absence of any command-line option, the default is the JML default: non-null by default.

Then, within a class, any use of a type name is non-null or nullable according to the default for that class, unless the type name is explicitly marked with one of the modifiers `non_null` or `nullable` or the annotations `@NonNull` or `@Nullable`.

In JML the use of `non_null` and `@NonNull` are equivalent, as are `nullable` and `@Nullable`.

The default rule for binaries is slightly different, as described in §4.9.4.

4.9.3 Nullness annotations for array declarations

Type annotations in Java are a bit complicated and less intuitive for array declarations, and somewhat backwards incompatible. Previously

```
/*@ non_null*/ String[] x;
```

meant that `x` was non-null and said nothing about the values in the array of Strings that `x` is a reference to.

With Java's type annotations, presuming for these examples that the default is *nullable*,

```
@NonNull String[] x;
```

means that `x` is a reference to a possibly-null array of non-null Strings.

```
String @NonNull [] x;
```

means that `x` is a reference to a non-null array of possibly-null String values. And

```
@NonNull String @NonNull [] x;
```

means `x` is a non-null reference to an array of non-null String values. For multi-dimensional arrays: In

```
String @NonNull [] [] x;
```

`x` is a non-null array of possibly null arrays of possibly-null Strings. In

```
String [] @NonNull [] x;
```

`x` is a possibly-null array of non-null arrays of possibly-null Strings. In

```
@NonNull String [] [] x;
```

`x` is a possibly-null array of possibly-null arrays of non-null Strings.

In the examples above, if the default is non-null rather than nullable, then all the levels of the type declaration are non-null except where explicitly annotated as `@Nullable`.

Settle default for array elements

Make sure of agreement on all the above with the JML RM

4.9.4 Nullness for binary classes

For binary (.class) files with no source code and no explicit specifications, a default set of specifications are presumed. These are by necessity (for soundness) presumed to be very conservative and slightly different than the defaults when source code is present. In this case,

- arguments to a method are presumed to be non-null
- return values are presumed to be possibly null
- any available fields are presumed to be possibly null

If these are too conservative, it is a simple matter to supply a `.jml` file that expresses the method's behavior more accurately. For soundness sake, however, the specification should still be conservatively underspecified where the behavior is not precisely known.

4.10 Arithmetic modes

4.10.1 Integer arithmetic

JML defines three arithmetic modes for integer arithmetic: java, safe, and bigint.

- In *java* math mode, all integer computations (negation, addition, subtraction, multiplication, division, modulo, casting, shifting) are performed precisely as in Java, in 2's complement fixed-bit-width arithmetic (either 32 or 64 bits). No warnings are given for overflow.
- In *safe* math mode, all operations still produce the same result as in Java, but OpenJML will issue an verification error if it cannot prove that overflows or underflows or shifts with a right-hand operand out of range do not occur. This is the JML default for analyzing Java code.
- In *bigint* math mode, all operations are performed using unlimited mathematical integers. This is the default for arithmetic in specifications.

The math mode for interpretation of Java code is set as a command-line option: either **--code-math=java** or **--code-math=safe**. OpenJML does not implement **--code-math=bigint** for Java code. The math mode for Java code can also be set using the modifiers `code_java_math` and `code_safe_math` on specific methods (overriding the global setting) or on a class or interface, applying to all methods in that class or interface (or in syntactically nested classes or interfaces).

The math mode for interpretation of JML specifications is set as a command-line option: either **--spec-math=java** or **--spec-math=safe** or **--spec-math=bigint**. The math mode for JML specifications can also be set using the modifiers `spec_java_math`, `spec_safe_math` and `spec_bigint_math` on specific methods (overriding the global setting) or on a class or interface, applying to all methods in that class or interface (or in syntactically nested classes or interfaces).

The checks performed for arithmetic overflow are *soft assertion* checks. That is, a warning is given (if in safe mode) that an overflow might happen, but the result of the operation is the same in any case and no assumption about the future program state is assumed. (cf. the discussion of hard and soft assertions in §9.1.1). For example, in

```
1 //@ requires i >= 0 && j >= 0;
2 void m(int i, int j) {
```

```
3   int k = i + j;
4   //@ assert i + j <= Integer.MAX_VALUE;
5 }
```

verification errors will be issued for both line 3, where an overflow might happen, and line 4, because no constraints have been put on `i` and `j` by the previous failure.

Arithmetic checks can be made hard with the command-line option **--arithmetic-failure=hard**. With this option enabled, the failure on line 3 adds the assumption that the overflow did not happen, that is, that `i + j <= Integer.MAX_VALUE`, and then the assert statement on line 4 passes.

4.10.2 Floating point arithmetic

Not yet implemented. It is expected that there will be alternate modes for floating-point arithmetic as well — performing all computations in precise IEEE floating point or using real arithmetic.

4.11 Integers and bit-vectors (--esc-bv option)

In Java (and other programming languages), integer values are sometimes used not as numbers but as sequences of bits. Perhaps each bit denotes some on or off value, with all of the bits packed into a single long or int or short or byte value. SMT solvers can reason both about numbers and about bit-vectors, but with some important caveats.

- A particular value is either a bit-vector or a number and cannot be converted from one to the other.
- Bit-vectors support all the arithmetic operations that numbers do, but numbers do not support bit-wise and, or, exclusive-or or shift operations. In some limited cases these operations can be emulated on numbers; for example, shifting by a literal integer amount can be replaced by multiplication or division.
- Proofs involving bit-vectors typically take much longer than on numbers.

Because of this last point, OpenJML encodes a method for SMT using numbers whenever possible and uses bit-vectors only when necessary because of

the choice of operations.

The **--esc-bv** command-line option controls the choice of using bit-vectors or not. Its values are **--esc-bv=true** to force using bit-vectors, **--esc-bv=false** to forbid it, and **--esc-bv=auto** (the default) to allow OpenJML to make the determination as described above.

Current OpenJML translates all integer values in a method as bit-vectors or all as numbers. This is overly constrained. SMT allows some quantities to be represented one way and some the other. Implementing such a mix is planned but not yet completed.

4.11.1 Specification inference

Precise specifications can be verbose and writing them can be time-consuming. If would be a productivity enhancement if straightforward specifications could be inferred automatically. There is a danger: specifications inferred from source code will likely have the same errors as the source code, and thus should be carefully reviewed.

JML itself does not define any inference. As a language it just defines the meaning of specifications and is mute on the question of the origin of those specifications. That is, it does not define any situations where specifications will be omitted because they will be accurately inferred. It only defines conservative defaults for missing specifications. It is up to tools like OpenJML to improve usability by inferring specifications where possible and appropriate.

This is a substantial topic and is the subject of §11.1.

Chapter 5

OpenJML Options

There are many options that control or modify the behavior of OpenJML. Some of these are inherited from the OpenJDK compiler on which OpenJML is based. The general behavior of options and properties is described in §4.3. All of the options are listed alphabetically in Tables 5.1 and 5.2. The options are then described in following subsections in functionally similar groupings or in other chapters relevant to their functionality.

Note that OpenJDK is migrating its options to generally use long-form names starting with two hyphens (--) and using lower-case, hyphen-separated words (dash-case). OpenJML traditionally used single-hyphen option names to match `javac`, with no single-letter abbreviations. ojml has now added and prefers the two-hyphen, dash-case spelling of its options, with the old spellings still supported as aliases.

Java (OpenJDK) options that are not relevant to OpenJML are only listed for completeness but not discussed here. See Java's documentation for more information on those [16].

For convenience these tables are replicated in the Appendix (Tables A.1 and A.2).

Options inherited from OpenJDK See the Java documentation for more detail	
@*<filename>*	read options from a file. *This is implemented only for Java options, not* OpenJML *options*
-Akey	options to pass to annotation processors
--add-modules *<modulelist>*	[§5.10] see Java documentation re modules
-bootclasspath *<path>* --boot-class-path *<path>*	See Java documentation
-cp *<path>* -classpath *<path>* --classpath *<path>*	[§4.2] location of input class files
-d *<directory>*	location of output class files
-deprecation	warn about use of deprecated features
--enable-preview	enables preview language features
-encoding *<encoding>*	character encoding used by source files
-endorsedirs *<dirs>*	see Java documentation
-extdirs *<dirs>*	see Java documentation
-g	generate debugging information
-h *<directory>*	location of generated header files
-? -help --help	[§5.5] output (Java and JML) help information
--help-extra	[§5.5] help about extra options
-implicit	whether or not to generate class files for implicitly referenced classes
-J*<flag>*	flags for the runtime system
--limit-modules *<modulelist>*	[§5.10] see Java documentation re modules
-m *<module>* --module *<module>*	[§5.10] see Java documentation re modules
--module-path *<path>*	[§5.10] see Java documentation re modules
--module-source-path *<path>*	[§5.10] see Java documentation re modules
--module-version *<version>*	[§5.10] see Java documentation re modules
-nowarn	[§5.5] show only errors, no warnings
-p *<path>*	[§5.10] like --module-path see Java documentation re modules
-parameters	see Java documentation
-proc	see Java documentation re annotation processing
-processor *<classes>*	see Java documentation re annotation processing
--processor-module-path *<path>*	see Java documentation re annotation processing
-processorpath *<path>* --processor-path *<path>*	where to find annotation processors
-profile	see Java documentation
--release *<release>*	target release for compilation
-s *<directory>*	location of output source files
-source *<release>* --source *<release>*	the Java version of source files

Options inherited from OpenJDK (cont.) See the Java documentation for more detail	
-sourcepath *<path>* --source-path *<path>*	[§4.2] location of source files
--system *<jdk>*	see Java documentation
-target *<release>* --target *<release>*	the Java version of the output class files
--upgrade-module-path *<path>*	[§5.10] see Java documentation re modules
-verbose	[§5.5] verbose output for Java compiler only, not OpenJML
-version --version	[§5.5] output (OpenJML) version
-X	[§5.5]Java non-standard extensions
-Werror	[§3.4]treat warnings as errors

Table 5.1: OpenJML options inherited from Java. See the text for more detail on each option.

Options specific to JML Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
--arithmetic-failure *<mode>*	[§4.10.1] sets the mode for arithmetic checks: hard, soft (the default) or quiet
[-]-check	[§5.3] typecheck only (**--command=check**)
--check-accessible -checkAccessible	[§7.3.4.1] whether to check accessible clauses (default: true)
[-]-check-feasibility *<list>* -checkFeasibility *<list>*	[§7.2] kinds of feasibility to check
[-]-check-specs-path -checkSpecsPath	[§4.2] warn about non-existent specs path entries
[-]-code-math *<mode>*	[§4.10] arithmetic mode for Java code (default: safe)
[-]-command *<action>*	[§5.3] which action to do: `check esc rac compile`, default is `check`
[-]-compile	[§5.3] typecheck JML but compile just the Java code (**--command=compile**)
[-]-counterexample -ce	[§7.3.5] show a counterexample for failed static checks
[-]-defaults *<list>*	enables various default behaviors TBD
[-]-determinism	EXPERIMENTAL: ???
--dir *<dir>*	[§5.4] argument is a folder or file; enables processing all .java files in a folder
--dirs	[§5.4] subsequent arguments are folders or files (until an argument is an option)
[-]-esc	[§5.3] do static checking (**--command=esc**)
--esc-bv	[§4.11] whether to use bit-vector arithmetic (default: auto)

Options specific to JML (cont.) Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
-escBV	
--esc-max-warnings *<n>* -escMaxWarnings	[§7.3.5] max number of verification errors to report in -esc
-escMaxWarningsPath	TBD? KEEP THIS?
[-]-exec *<file>*	[§7.3.2] file path to prover executable
[-]-exclude *<patterns>*	[§7.3.3] paths to exclude from verification
[-]-extensions *<classes>*	[§10] comma-separated list of extensions classes and packages
[-]-inline-function-literal	EXPERIMENTAL ?
-java	[§5.3] use the native OpenJDK tool
-jml	[§5.3] process JML constructs
-jmldebug	[§5.5] very verbose output (includes -progress) (**--verboseness=4**)
[-]-jmltesting	changes some behavior for testing (default: false)
[-]-jmlverbose	[§5.5] JML-specific verbose output (**--verboseness=3**)
[-]-keys	[§4.5] define keys for optional annotations
[-]-lang *<language>*	[§9] the JML variant to use
[-]-logic *<name>*	[§7.3.2] name of SMT logic to use (default: ALL)
[-]-method *<patterns>*	[§7.3.3] methods to include in verification
--nonnull-by-default -nonnullByDefault	[§5.4] values are not null by default
[-]-normal	[§5.5] only outputs errors; no pother progress information (**--verboseness=1**)
--nullable-by-default -nullableByDefault	[§5.4] values may be null by default
[-]-osname *<name>*	[§7.3.2] Operating System name to use in selecting prover (default: "" (auto), or one of `macos`, `linux`, `windows`)
[-]-progress	[§5.5] outputs errors, warnings, progress and summary information (**--verboseness=2**)
[-]-properties *<file>*	[§4.3] property file to read (value required)
[-]-prover *<name>*	[§7.3.2] prover to use (default: z3-4.3)
-purityCheck	[§5.4] check for purity
[-]-quiet	[§5.5] no informational output (**--verboseness=0**)
[-]-rac	[§5.3] compile runtime assertion checks (**--command=rac**)
--rac-check-assumptions -racCheckAssumptions	[§8.3.4] enables (default on) checking assume statements as if they were asserts
--rac-compile-to-java-assert -racCompileToJavaAssert	[§8.3.6] compile RAC checks using Java asserts
--rac-java-checks -racJavaChecks	[§8.3.5] enables (default on) performing JML checking of violated Java features
-racMissingModelFieldRepSource	TBD
-racMissingModelFieldRepBinary	TBD

Options specific to JML (cont.) Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
--rac-precondition-entry -racPreconditionEntry	TBD
--rac-show-source -racShowSource	[§8.3.3] includes source location in RAC assertion failure messages
[-]-require-white-space	[§6.1] whether white space is required after an @ (default: false)
[-]-show	[§5.5] prints the details of source transformation (default: false)
--show-not-executable -showNotExecutable	[§8.3.1] warn about features not executable, in --rac operations (default: TBD)
--show-not-implemented -showNotImplemented	[§5.4] warn about features not implemented (default: TBD)
--silent	[§5.5] turns off all (error, warning, informational) output except the error code (**--verboseness=-1**)
--show-skipped -skipped	[§7.3.3] show methods whose proofs are skipped (default: true)
--smt *filename*	[§7.3.9] where to write generated SMT files (for off-line use or inspection)
[-]-solver-seed	[§7.3.9] seed to pass on to the SMT solver (default: 0 - no seed)
[-]-spec-math <*mode*>	[§4.10] arithmetic mode for specifications (default: bigint)
--specs-path -specspath	[§4.2] location of specs files
[-]-split	[§9.1.5] splits proof of method into sections
--stop-if-parse-errors -stopIfParseErrors	[§6.1] stop if there are any parse errors (don't do type checking or verification attempts)
-staticInitWarning	TBD
[-]-subexpressions	[§7.3.5] show subexpression detail for failed static checks (default: false)
[-]-timeout <*seconds*	[§7.3.9] timeout for individual prover attempts (default: TBD)
[-]-trace	[§7.3.5] show a trace for failed static checks (default: false)
[-]-triggers	enable SMT triggers (default: true)
-typeQuants	TBD
[-]-verboseness <*n*>	[§5.5] level of verboseness (0=quiet .. 4=jmldebug) (default: 1, -normal)
[-]-verify-exit <*n*>	[§7.3.9] exit code for verification errors (default: 6)
[-]-warn <*list*>	[§5.8] comma-separated list of warning keys (default: no keys)

Table 5.2: OpenJML options. See the text for more detail on each option.

5.1 General rules about options

- The command-line consists of the path to the executable followed by space-separated arguments. Arguments that contain spaces should be enclosed in double-quotes. The shell interpreter and the OS being run will dictate other properties of the command-line, such as when variables are substituted, when filename expansion is performed, and how file-system paths are written.
- The arguments themselves are either (relative or absolute) paths to files or options. An option may by followed by a value (if it requires a value), which is then either the next argument in the command-line or combined with the option name by an = character. Relative paths are relative with respect to the current working directory (as given by `pwd`, for example).
- Options begin with an initial hyphen character. It is now more common to have long option names begin with two hyphens and abbreviated names begin with one (as in --**help** and -**h**) and some javac options do have alternative double-hyphen version. OpenJML has also introduced two-hyphen option names, though most older one-hyphen names are still retained (though discouraged), as shown in Table 5.2.
- If an option appears more than once, then the values designated by later (to the right) appearances override earlier appearances; options that are not listed have default values.
- Default values can be set by properties and environment variables (cf. §4.3), otherwise a built-in value is used.
- Options may have boolean or string values, though string values may be constrained to a specific format, such as a numeral.
- A boolean option (e.g. --**xyz**) is set to true by either
  --**xyz** or --**xyz=true**,
  and set to false by either
  --**no-xyz** or --**xyz=false**;
  --**xyz=** resets the option to its built-in default.
- A string option is required to have a value, which is specified either by --**xyz=*value*** (preferably for JML options) or --**xyz *value*** (without an = connector). Only some double-hyphen Java options may use the = form. The

form --**xyz=** resets the option to its built-in default.

5.2 Options: Operational modes

These operational modes are mutually exclusive.

- -**jml** (default) : use the OpenJML implementation to process the listed files, including embedded JML comments and any corresponding `.jml` files
- -**no-jml**: uses the OpenJML implementation to type-check and possibly compile the listed files, but ignores all JML annotations in those files
- -**java**: processes the command-line options and files using only OpenJDK functionality. No OpenJML functionality is invoked and no other OpenJML options are allowed. It must be the first option.

5.3 Options: JML tools

The following mutually exclusive options determine which OpenJML tool is applied to the input files. They presume that the `-jml` mode is in effect.

- --**command** <*tool*> : initiates the given function; the value of <*tool*> may be one of **check**, **esc**, **rac**, **compile**, **doc**. The default is to use the OpenJML tool to do only typechecking of Java and JML in the source files (**check**).
- --**check** : causes OpenJML to do only type-checking of the Java and JML in the input files (alias for --**command=check**)
- –**compile** : causes OpenJML to do JML type-checking (as with --**check**), but then compiles the Java code without any runtime-checking (a rarely used option) (alias for --**command=compile**)
- --**esc** : causes OpenJML to do (type-checking and) static checking of the JML specifications against the implementations in the input files (alias for --**command=esc**)
- --**rac** : compiles the given Java files as OpenJDK would do, but with JML checks included for checking at runtime (alias for --**command=rac**)
- --**doc** : executes javadoc but adds JML specifications into the javadoc output files (alias for --**command=doc**) *Not yet implemented.*

5.4 Options: OpenJML options applicable to all OpenJML tools

- **--dir** *<folder>* : abbreviation for listing on the command-line all of the `.java` files in the given folder and its subfolders (recursively); if the argument is a file, use it as is. A warning is issued if the given path does not exist.
- **--dirs** : treat all subsequent command-line arguments as if each were the argument to **--dir**, until reaching an argument that begins with a hyphen character. Note that this sequence of arguments may contain arguments with wild-card characters that are expanded by the shell. For example, `--dirs A*.java` would expand to all the java files in the current folder that begin with 'A', and would work as expected. It is OK if this option has no values — that is, if the very next command-line argument begins with a hyphen. This might occur, for example, if in `--dirs A*.java` there were no files matching the given pattern.
- **--specs-path** *<path>* : defines the specifications path, cf. §4.2, which is analogous to classpaths and sourcepaths
- **--keys** *<keys>* : the argument is a comma-separated list of conditional annotation keys (cf. the JML Reference Manual), used to conditionally enable or disable designated annotations (cf. §4.5)
- **--show-not-implemented** : emits warnings about JML features that are ignored because they are not implemented; the default is disabled (silently ignoring such features).
- **--nullable-by-default** : sets the global default to be that all declarations are implicitly `@Nullable`, if they are not explicitly declared `@NonNull` (cf. §4.9)
- **--nonnull-by-default** : sets the global default to be that all declarations are implicitly `@NonNull` (the default), if not explicitly declared `@Nullable` (cf. §4.9)
- **--check-specs-path** : if enabled, checks that each element (directory or jar files) of the *specspath* actually exists; if disabled (with **--no-check-specs-path**), non-existent entries are silently ignored (default: enabled)

5.5 Options: JML Information and debugging

These options print summary information and immediately exit (despite the presence of other command-line arguments):

- -**?** , -**help**, --**help** : prints out help information about the command-line options
- --**version** : prints out the version of the OpenJML tool software
- -**X**, --**help-extra** : Java option to print out help about advanced or experimental options

The following options provide different levels of verboseness. If more than one is specified, the last one present overrides earlier ones.

- --**silent** : only an exit code
- --**quiet** : no informational output, only errors and warnings; warnings can be omitted using -**nowarn** along with --**quiet**
- --**normal** : (default) some informational output, in addition to errors and warnings
- --**progress** : prints out summary information as individual files are processed and proofs are attempted (includes --**normal**)
- --**verbose** : prints out verbose information about the Java processing in OpenJDK (does not include other OpenJML information)
- --**jmlverbose** : prints out verbose information about the JML processing (includes --**verbose** and --**progress**)
- --**jmldebug** : prints out (voluminous) debugging information (includes --**jmlverbose**)
- --**verboseness** <*int*> : sets the verboseness level to a value from -1 .. 4, corresponding to --**silent**, --**quiet**, --**normal**, --**progress**, --**jmlverbose**, --**jmldebug**
- -**nowarn** : this Java option turns off printing of warnings, leaving only errors and verification failures

Other debugging options:

- --**show** : prints out rewritten versions of the Java program files for informational and debugging purposes. It is generally useful to confine this output to a single method using the --**method**=*methodname* option. There are four parts to this output.
  - --**show** prints all four
  - --**show=program** prints the original program from its AST, after parsing and type resolution
  - --**show=translated** prints each method after JML statements have been translated into Java, for either ESC or RAC
  - --**show=bb** prints each selected method after basic-block transformations (ESC only)
  - --**show=smt** prints the smt commands as sent to the solver (if only this output is needed, the --**smt** option is likely more convenient) (ESC only)

  A comma-separated list of a selection of the four identifiers may also be used. Note that this output is quite lengthy.

An option used primarily for testing:

- -**jmltesting** : reduces the output so that test output is more stable
  - no timing or prover identification information is output
  - the verification success/failure summary is not output (as in --**no-show-summary**)
  - does not use the verification failure exit code (§3.4) (only until the test output can be updated)
  - uses 'warning' instead of 'verify' in verification assertion failure messages (only until the test output can be updated)
  - does not show location back-pointer information in 'Associated declaration' messages (only until the test output can be updated)
  - in RAC, some location information is suppressed

5.6 Java Options: Version of Java language or class files

- --**source** <*level*> : this option specifies the Java version of the source files, with values of `4`, ..., `17`, ... This controls whether some syntax features

are permitted. The default is the most recent version of Java (currently 17).

- **--target** <*level*> : this option specifies the Java version of the output class files (for compilation or RAC)

5.7 Java Options: Other Java compiler options applicable to OpenJML

All the OpenJDK compiler options apply to OpenJML as well. The most commonly used or important OpenJDK options are listed here.

These options control where output is written:

- **-d** <*dir*> : specifies the directory in which output class files are placed; the directory must already exist
- **-s** <*dir*> : specifies the directory in which output source files are placed; such as those produced by annotation processors; the directory must already exist

These are Java options relevant to OpenJML whose meaning is unchanged in OpenJML.

- **--class-path** or **-cp** or **-classpath**: the parameter gives the Java classpath to use to find referenced classes whose source files are not on the command-line (cf. §4.2)
- **--source-path** or **-sourcepath**: the parameter gives the sequence of directories in which to find source files of referenced classes that are not listed on the command-line (cf. §4.2)
- **-deprecation**: enables warnings about the use of deprecated features (applies to deprecated JML features as well)
- **-nowarn**: shuts off all compiler warnings, but not verification failures or Java and JML language errors
- **-Werror**: turns all warnings into errors, including compiler, JML typechecking and JML verification failures
- **-verbose**: turn on Java verbose output (does not control JML output)
- **-Xprefer:source** or **-Xprefer:newer**: when both a .java and a .class file are present, whether to choose the .java (source) file or the file that has the more recent modification time [TBD - check that this works]

Other Java options, whose meaning and use is unchanged from javac (and rarely used by OpenJML:

- **@*<filename>*** : reads the contents of *<filename>* as a sequence of command-line arguments (options, arguments and files), but Java options only
- **-Akey**
- **-bootclasspath**
- **-encoding**
- **-endorsedirs**
- **-extdirs**
- **-g**
- **-implicit**
- **-J**
- **-X**... : Java's extended options

5.8 Control of lint-like warnings

OpenJML issues a number of suggestions about style and possible erroneous, though not explicitly illegal use of JML. The **--warn** option enables control of such warnings.

These warnings are grouped into categories, described below; each category can be enabled or disabled individually. Each category has its own default as to whether it is enabled or disabled by default.

- **--warn=all** — enable all warning categories
- **--warn=** — reset all warning categories to their defaults
- **--warn=default** — reset all warning categories to their defaults
- **--warn=none** — disable all warning categories
- **--no-warn=all** — disable all warning categories
- **--warn=***list* — enable the given categories (leaving other categories unchanged), where *list* is a comma-separated list of category names
- **--no-warn=***list* — disable the given categories (leaving other categories unchanged), where *list* is a comma-separated list of category names

This list and option are under development

5.9 Java options related to annotation processing

Java has an annotation processing facility, affected by the options below. JML and OpenJML do nothing with annotation processing. It has not been tested whether OpenJML works in conjunction with annotation processing.

- **-proc**
- **-processor**
- **-processorpath**

5.10 Java options related to modules

Java 11 introduced modules to the Java language for the purpose of controlling access to code more tightly than the Java visibility mechanism does. No interaction between JML and modules has been defined in JML or implemented in OpenJML. Generally speaking, programs using JML should just use the default, unnamed module.

- **--add-module**
- **--limit-modules** *<modulelist>*
- **-m** *<module>*
- **--module** *<module>*
- **--module-path** *<path>*
- **--module-source-path** *<path>*
- **--module-version** *<version>*
- **-p** *<path>*
- **--upgrade-module-path**

The above options are all Java options for handling modules, as of Java 11. JML does nothing about modules per se, leaving all visibility checking to OpenJDK.

Check that the option lists are comprehensive, and up to date with Java 17

Chapter 6

OpenJML tools — Parsing and Type-checking

6.1 Parsing

OpenJML parses the `.java` files listed on the command-line, finds any corresponding `.jml` files, and then also finds the files corresponding to classes mentioned in files already parsed. If a class has a `.class` file on the class-path then it and any corresponding `.jml` file are read; if there is no already compiled `.class` file (or the source file is newer or preferred, cf. the **-Xprefer** OpenJDK option) then OpenJML finds and parses the source and specification file for the class.

Parsing is affected by these options:

- the classpath, sourcepath and specspath (§4.2)
- the **--stop-if-parse-errors** causes the tool to stop after parsing files if any parse errors are found. This is a fail-fast practice, rather than proceeding with typechecking as much as possible to see what other errors there might be. In any case, no verification attempts will be tried if there are any parsing or typechecking errors.
- the **--require-white-space** option. If this option is enabled (disabled by default) then a comment beginning with `//@` or `/*@` is only considered to be JML if there is white space after the (sequence of) `@` symbol. This option is disabled by default but can be useful when incorporating source

files that had Java annotations (e.g. `@Override`) that were commented out to produce `//@Override`. Using this option avoids having non-JML comments like these interpreted as erroneous JML comments.

6.2 Type-checking JML specifications

The type-checking phase includes all of OpenJDK's name and type attribution for Java; OpenJML adds type-checking of any JML annotation text and any `.jml` files. OpenJML also ensures that the `.jml` files match the contents of the Java `.class` or `.java` files.

A set of Java files with JML annotations is parsed and type-checked with the command

```
openjml -{}-check \textit{options} \textit{files}
```

or

```
openjml \textit{options} \textit{files}
```

since --**check** is the default action. Any `.jml` files are checked when the associated `.java` file is checked. Only `.java` files either listed on the command-line or contained in folders listed on the command-line are certain to be checked. Some checking of other files may be performed where references are made to classes or methods in those non-listed files.

6.3 Command-line options for type-checking

The following command line options are particularly relevant to type-checking.

- –**purity-check** : turns on (the default) purity checking of library methods. Using Java library methods in specifications before specifications are written for the called method usually provokes a complaint that the library method is not pure and may not be used in a specification. The –**no-purity-checking** can be used temporarily suppress such type-checking errors while specifications are being written. (This option is slated for deprecation.)

Chapter 7

OpenJML tools — Static Deductive Verification (ESC)

Type-checking is performed automatically prior to ESC (Extended Static Checking). Thus ESC also depends on the information described in Chapters 3, 5 and 6, particularly including the command-line options relevant to type-checking and the discussion of class, source, and specification paths in §4.2.

7.1 Results of the static verification tool

The ESC tool operates on a method at a time. Which methods are considered in a given execution of OpenJML are determined by options (cf. §7.3.3). The ESC tool will result in one of four outcomes:

- It issues one or more verification failure messages.
- It finds no verification failures.
- It exhausts memory resources or allotted time.
- It encounters some internal bug.

These scenarios are discussed in the following subsections.

7.1.1 Finding verification faults

A run of OpenJML with **--esc** may find one or more static checking warnings. Current OpenJML will find all the static check problems it can within a method. However, the **--max-esc-warnings** option can limit the search to just one warning, or it can keep searching until a certain number of warnings are found, or until no additional warnings can be found. If the goal is simply to determine whether there are any faults, stopping at just one will save time; if the goal is to find and fix all the faults, it may be convenient to search until no more can be found. If there are multiple faults, the order in which they are found is non-deterministic.

The static warnings found are grouped into various categories. For example if a method is called but the method's precondition cannot be proved to hold, then a `Precondition` warning is reported. An explicit JML `assert` that cannot be proved true, will result in an `Assert` warning. The various categories of warnings are listed in Appendix B.

Note that static warnings are reported if the tool cannot prove that the associated verification condition is satisfied. It may be that the verification condition is indeed valid, but the tool simply is unable to prove it.

Give an example

7.1.2 Checking feasibility

A run of OpenJML with **--esc** may find no warnings through static checking. In this case, the tool can run additional checks to be sure the program is *feasible*, that is, that the specifications and the implementation actually permit execution of the program. By default, OpenJML does not do feasibility checking because it can be misleading or time-consuming; however a careful verification process will do some level of feasibility checking before considering a verification successful. Feasibility checking is discussed in more detail in §7.2.

7.1.3 Timeouts and memory-outs

The underlying SMT solvers may report a time-out or memory exhaustion. One option is to increase the time out limit (with the **--timeout** option). An alternate recourse in this situation is to attempt to simplify the implementation or the

specification. A time-out option to OpenJML is passed through to the underlying SMT solver for it to interpret according to its own implementation, so the user can do some experimentation. When running static checking on a whole group of methods, it is useful to use a somewhat short time-out value, so that particularly difficult methods do not unduly delay obtaining results for other methods.

The value of the timeout option is the number of seconds to which to limit the proof attempt, for each method or method split or feasibility check, individually.

If OpenJML ends by exhausting memory, it is generally a problem with the solver. There is currently no control over the memory available to the SMT solver (aside from finding a larger computer).

7.1.4 Bugs

Despite the author's continuing efforts, there still remain bugs and limitations in OpenJML. If you encounter any, please report them with as much information as possible, via the OpenJML GitHub project:

https://www.github.com/OpenJML/OpenJML/issues .

A useful bug report includes all the source code required to reproduce the problem, the operating system being used, the version of Java and OpenJML; the most useful reports will pare down the source code to a minimum amount that still provokes the error.

7.2 Checking feasibility: --check-feasibility

--check-feasibility *where*: checks feasibility of the program at various points as described below. The default is `none`.

Deductive verification typically asks the question: are there any legal inputs that would render an implicit or explicit assertion false? A second question is: are there any legal inputs that cause execution to reach a given point in the program? That is, is the execution path to that point in the program *feasible*?

The question of feasibility can be important for several reasons.

- If there is indeed some infeasible execution path, then any assertions on that path will not be checked. Then a verification attempt can be success-

ful (no verification errors reported), when in fact that success is because *there was nothing to check* (because that or maybe all execution paths are infeasible). Thus after a successful verification attempt it can be prudent to check feasibility.

- If there are contradictory assumptions (e.g., assume statements or preconditions or invariants) then any point after those assumptions will not be feasible. For example

```
// openjml --esc --check-feasibility=exit T_Feasibility1.java
public class T_Feasibility1 {

  //@ requires i < 0;
  //@ ensures \result > 0;
  public int m(int i) {
    //@ assume i > 0;
    return i;
  }
}
```

produces

```
T_Feasibility1.java:6: verify: There is no feasible path to program point
    at program exit in method T_Feasibility1.m(int)
  public int m(int i) {
             ^
1 verification failure
```

- When method A calls method B, the verification of method A relies on correct specifications for method B. Consider this example:

```
// openjml --esc --check-feasibility=call T_Feasibility4.java
abstract class A {
    public int kk;
    //@ ensures kk == \old(kk) + 1;
    //@ pure // faulty spec
    abstract public void mm();
}
abstract public class T_Feasibility4 extends A {
    //@ requires i > 0;
    public void m(int i) {
        mm();
    }
}
```

Verification without checking feasibility reports no errors. However, when feasibility is checked, a problem is reported with the call of ‘mm()‘.

```
T_Feasibility4.java:11: verify: There is no feasible path to program point
    after call in method T_Feasibility4.m(int)
        mm();
          ^
```

```
1 verification failure
```

The problem here is that the specs of ‘mm()‘ say that the method is ‘pure‘, meaning that it changes nothing, but the ensures clause says that ‘k‘ is incremented. This contradiction results in stopping any verification after the method call. The feasibility check indeed finds this problem. This example points out the necessity of verifying all methods used in a program before the program can be considered verified. This is particularly relevant to library methods. These may well have specifications, but a typical client of the library will be forced to trust these specifications and will not have the source code to even attempt a verification of the library methods the client uses.

- Some branches of the code may be *dead*, that is, are never executed. In fact sometimes one may wish to prove that a branch, such as an error reporting or recovery branch, will not be executed. Feasibility checking can assist in detection of dead code.

All the various places that OpenJML implements feasibility checking are enumerated below. But first, some caveats are in order.

- Feasibility checking can be time-consuming and especially so if the path in question is *not* feasible. Accordingly, feasibility checking is off by default.
- Feasibility checking only says that some input combination will reach the given program point, not whether all the combinations you expect will reach that point. For example, if a program has assumptions `i<= 0` and `i >= 0`, it will still be feasible for `x == 0`, but that may not be the programmer's intent.
- If method A calls method B and method B is underspecified, then an execution path may be considered to be feasible, when in reality it is not. Remember that when checking method A, only the specifications of B are considered. Look at this example:

```
// openjml --esc --check-feasibility=reachable T_Feasibility2.java
public class T_Feasibility2 {

  //@ requires i >= 0;
  public void m(int i) {
    int j = abs(i);
    if (i != j) {
      // Should never get here!
      //@ reachable
```

```
10      }
11    }
12
13    //@ requires i != Integer.MIN_VALUE;
14    //@ ensures \result >= 0;
15    public static int abs(int i) {
16      return i < 0 ? -i : i;
17    }
18  }
```

The command stated at the top of the example checks whether it is possible to reach the ‘reachable‘ statement in the program. Indeed, the check runs without complaint, meaning that the program point is indeed reachable. Given that for positive numbers, the ‘abs‘ method should just return its input, how can this be? Well,. in verifying method ‘m‘ all we see is the specification of ‘abs‘. That specification is *underspecified*. It only says that the output is non-negative, not that it is equal to the input or its negation. Replacing the ‘reachable‘ statement with an ‘unreachable‘ statement helps us do some debugging:

```
1  // openjml --esc T_Feasibility3.java
2  public class T_Feasibility3 {
3
4    //@ requires i >= 0;
5    public void m(int i) {
6      int j = abs(i);
7      //@ show i, j;
8      if (i != j) {
9        // Should never get here!
10       //@ unreachable
11     }
12   }
13
14   //@ requires i != Integer.MIN_VALUE;
15   //@ ensures \result >= 0;
16   public static int abs(int i) {
17     return i < 0 ? -i : i;
18   }
19 }
```

produces

```
1  T_Feasibility3.java:7: verify: Show statement expression i has value 1
2      //@ show i, j;
3                ^
4  T_Feasibility3.java:7: verify: Show statement expression j has value 2
5      //@ show i, j;
6                   ^
7  T_Feasibility3.java:10: verify: The prover cannot establish an assertion (
       Unreachable) in method m
8        //@ unreachable
9            ^
```

```
10 3 verification failures
```

which shows that that the verifier thinks that ‘i‘ and ‘j‘ can be different (the specific values of ‘i‘ and ‘j‘ may be different from run to run).

So feasibility checking can be useful if these caveats are kept in mind. Feasibility checking is disabled by default and is enabled with the **--check-feasibility** option. The argument of that option is a comma-separated list of location identifiers, listed below. In addition there are some common combinations:

- **none** – turns off any feasibility checking
- **basic** – turns on just precondition, assert, assume, reachable, exit, halt, and spec
- **all** – turns on everything
- **debug** – just for debugging of OpenJML itself

Here are the possible places that can be checked:

- **reachable** – all points in the method explicitly marked with a ‘//@ reachable;‘ statement
- **precondition** – at the beginning of the method body; checks whether there are contradictions in the preconditions and invariants
- **assert** – just before each explicit assert statement; if the execution path to the assertion is not feasible, the assertion will never be checked
- **assume** – just after each explicit assume statement; if the execution path is not feasible, there is something wrong with the predicate being assumed (or something wrong before it)
- **return** – is every return statement feasible (after computing the return value)
- **throw** – is every throw statement feasible (after computing the throw expression)
- **if** – are both branches of the if condition feasible
- **switch** – are all branches of a switch statement feasible
- **catch** – at the beginning of each catch block
- **finally** – at the beginning of each finally block
- **spec** – at the end of every statement spec block
- **call** – after any call
- **halt** – at each halt statement
- **loopcondition** – at the beginning of the loop body
- **loopexit** – on the exit branch after testing the loop condition

- **exit** - is it possible to exit the program (normally or with an exception)

Need examples

7.3 Options specific to static checking

7.3.1 Controlling nullness

- --**nullable-by-default**: sets the global default to be that all variable, field, method parameter, and method return type declarations are implicitly `@Nullable`
- --**nonnull-by-default**: sets the global default to be that all variable, field, method parameter, and method return type declarations are implicitly `@NonNull` (the default)

Nullness control is discussed more fully in §4.9.

7.3.2 Choosing the solver used to check (--prover, --exec)

OpenJML uses SMT solvers to check all the conditions that are implied by the program and its specifications. In principle, any solver compliant with SMT-LIB-v.2.5[7] can be used. In practice, there are some limitations.

First, only a few solvers support the range of SMT-LIB logics that are used by OpenJML. Software verification naturally uses quantified expression, models of arrays, bit-vectors, mathematical integers and reals with non-linear operations, strings, sets, and sequences; in short, any well-defined mathematical object useful in describing how a piece of software works would be helpful. Some SMT solvers support just one logic, such as quantifier-free bit-vectors; a few support every logic defined in SMT-LIB, which is only a subset of the list above.

Second, the existing SMT solvers do not completely support SMT-LIB-v2.5. Consequently there is an adapter library, jSMTLIB[7], that translates standard SMT-LIB to an input suitable for the SMT solvers it supports. Further then, a new version of an SMT solver must be supported by jSMTLIB before it can be used. jSMTLIB does have a generic path for a fully-compliant solver.

Third, the various solvers differ in their capabilities. Some are faster or more reliable than others, perhaps just for particular logics. So it is useful to try different

solvers on non-trivial proof problems.

- --**prover** ***prover***: the name of the prover to use: one of
  - `z3_4_3`: description of versions here and for each item
  - `z3_4_5`
  - `cvc4`
  - `yices2`
  - [TBD: expand list]
  - What to say for a compliant SMT solver
- --**exec** ***path***: the absolute path to the executable corresponding to the given prover

Solvers typically have different executables for different operating systems. OpenJML automatically chooses the correct solver based on its detection of the operating system on which it is running (using the Java `os.name` system property). That determination can be overridden using the --**os-name** option, which recognizes the values `macos`, `linux`, and `windows`. In an OpenJML distribution, the different solvers are placed in subfolders named `Solvers-`*OS* for each supported *OS* name.

One other option related to solvers is –-**logic**, whose value is the SMT logic to use. Current SMT solvers select a SMT logic automatically, typically as all but just those logics needed to process the given SMT input. Thus this option is rarely needed and may well fail if actually tried. It may be removed in the near future.

7.3.3 Choosing what to check (--method, --exclude)

The default behavior is to check each method in each file and folder listed on the command-line (or selected in the GUI). The set of methods checked can be constrained by these options. In particular the --**method** option is often used to constrain checking to a single method while that method or its specifications are being debugged.

- --**method** <*methodlist*> : a semicolon-separated list of method names to check (default is all methods in all listed classes)
- --**exclude** <*methodlist*> : a semicolon-separated list of method names to exclude from checking (default: no methods are excluded)
- `skipesc` : a modifier on a class or method that indicates not to verify that method or methods in that class (cf. §9.2.1)

Table 7.1: Effect of --**method** and --**exclude**

--method option	--exclude option	result
no option present or match	none or no match	checked
option present but no match	none or no match	skipped
-	match	skipped

The --**method** and --**exclude** options interact as shown in Table 7.1; in summary, --**exclude** overrides --**method**.

- If there are multiple instances of --**method** options, only the last one applies, as is the rule for all options. The same applies to the --**exclude** option. To specify multiple methods or exclude rules, use one option with a semicolon-separated list of strings.
- If a method is skipped because of these rules, then any classes or methods within the skipped method are also skipped.
- Despite the --**method** option, any method or type annotated with `@SkipEsc` or `skipesc` is skipped
- The name of a constructor is the name of the class.
- There is no way to name anonymous classes or lambda functions in order to check or skip them.
- The list of strings to match is *semicolon*-separated rather than comma-separated because method signatures can contain commas. If multiple entires are separated by semicolons, you will likely have to quote the whole option to avoid the shell considering the semicolon the end of the command.

The --**show-skipped** option (§7.3.3) controls output about which methods are being skipped for verification. Using this option (which is on by default) prevents silently forgetting that some method is not being proved.

Matching rules. The argument of the --**method** and --**exclude** options is a semicolon-separated set of strings. A method *matches* if any one of the individual strings matches the name of the method. A match occurs if anyone of the following is true:

- the string is the simple name of the method

- the string is the fully-qualified name of the method
- the string is the fully-qualified signature of the method, with the arguments represented just by their fully-qualified types (and no white space)
- the string, interpreted as a regular expression (in the sense of `java.util.regex.Pattern`) matches the fully-qualified signature of the method

For example, the method `mypackage.MyClass.mymethod(Integer i, int j)` is matched by any of the following:

- `mymethod`
- `mypackage.MyClass.mymethod`
- `mypackage.MyClass.mymethod(java.lang.Integer,int)`
- `*MyClass*`

7.3.4 Control over what is checked

It is helpful sometimes to suppress some kinds of checks in order to focus on other problems or because a specification is still in development.

7.3.4.1 Checking accessible (reads) clauses: --check-accessible

The accessible clauses state what memory locations a method may read. When writing specifications they are often left until later in the process to be written; often they are just left as `accessible \everything`.

So it is sometimes useful to disable checking these clauses: **--no-check-accessible** does so. The check is enabled with **--check-accessible**, which is the default.

purity checking

staticInitWarning

7.3.5 Detail about the proof result

When OpenJML+SMT is unable to validate an assertion, it can be difficult to debug the problem: the problem can be either an insufficiently capable solver or mismatched specifications and implementation. The following options provide some tools to help understand the proof results.

- --**esc-max-warnings** ***int***: the maximum number of assertion violations to look for; the argument is either a positive integer or `All` (or equivalently `all`, default is `All`)
- --**trace**: prints out a counterexample trace for each failed assert
- --**subexpressions**: prints out a counterexample trace with model values for each subexpression
- --**counterexample** or -**ce**: prints out counterexample information

Provide more information and examples

7.3.6 Dividing up the proof: --split

The `split` statement (§9.1.5) enables splitting up the usually single, large verification condition for a method into sections that may be more manageable. The --**split** option controls which splits of the verification condition are attempted. Splitting proofs and this option are described in §9.1.5.

7.3.7 Controlling output

ESC can take a while to run if operating on a large set of software. It is useful then to have good progress reporting and to control the output produced. The basic controls are the level of verbosity, in particular the --**progress** setting and the options described in the previous subsection (§7.3.5).

On a first run through a large set of data, it is helpful to use the following set of options:

- --**progress** : so that the starting and completing each method is reported; these delineations also serve to associate warning and error reports with the method that produced them
- --**esc-max-warnings=1** : just one warning per method saves time and is enough to tell whether further work will be needed. Allow a higher limit when detailed analysis is being performed on just one or a few methods.
- --**check-feasibility=none** : (which is the default)
- Do not request tracing or counterexample information : this information is most helpful during debugging of single methods; in runs over many methods it just adds (voluminous) information that makes the output more difficult to understand

Such an initial run gives an overall understanding of where there are proof problems. Subsequent analysis can then be concentrated on problem points.

7.3.8 Options affecting the internal encoding

There are a variety of ways to encode Java and JML source code into logical expressions. Indeed this is an ongoing area of research. OpenJML may implement more than one technique for some aspect of encoding and define an option to allow selecting between them and experimenting with their effectiveness.

Such options may be transient, only lasting until their effectiveness is established or disproved.

Duplicated from an earlier section

- **--esc-bv** *mode* : this option controls whether fixed-bit-width integer operations are encoded as mathematical integer operations or as operations on bit-vectors. Reasoning about bit-vectors can take much longer than reasoning about integers. However, some operations, such as bit-wise and, or, and exclusive-or can only be represented in bit-vectors. This option has three choices:
  - `false` : use mathematical integers always, with an error if that is not possible
  - `auto` (the default) : let OpenJML choose integer encoding if possible, otherwise use bit-vector encoding
  - `true` : always use bit-vector encoding

7.3.9 Miscellaneous options

- **--solver-seed** *n* : solvers typically are non-deterministic in their approach to searching for a proof or counterexample, basing some search decisions on some internal pseudo-random number generator. This property leads to runtimes and even proof success varying from one run to another. That nondeterminism can be reduced by specifying the seed the solver should start with; OpenJML passes the value of this option to the backend solver. The effectiveness in reducing nondeterminism varies. A value of `0` means not to set a seed.

- --**timeout** *secs* : Some proofs take a long time. In production work it is advisable to set a timeout. This option takes an integer number of seconds and passes it on to the solver. (OpenJML does not itself impose a timeout; it is up to the solver to do so correctly. You can also use a shell wrapper command such as the Linux/MacOS `timeout` shell command to limit the runtime of OpenJML itself.)
- --**verify-exit** *n* : By default if the input to OpenJML has a verification failure (but no other errors), OpenJML will exit with an exit code of `6` (cf. §3.4). This option allows you to set that value to an integer in the range `0-6`, that is, to set it to be the same as a different kind of error or to be `0` and not an error at all.
- --**smt** *file-pattern* : This option gives the names of a file into which will be written the generated SMT-LIB commands that are sent to the SMT solver as the proof attempt for a method. If a simple filename is used, then the file will be overwritten for each method. So that option is most useful when a single method is being verified. Alternatively, on can include in the *file-pattern* the sequence `%_`, which will be replaced by the name of the method, or the sequence `%%`, which will be replaced by a fully-qualified method signature (which is unique within a set of classes). If the pattern is an empty string, a fixed name (`out.smt2`) is used. The default (--**no-smt**) is not to write any output file.

Chapter 8

OpenJML tools — Runtime Assertion Checking (RAC)

In Runtime Assertion Checking, a program is compiled to carry out its normal function, except that various assertions are compiled in and checked during the program's execution. If any assertions are found to be false, some error indication is emitted. In the case of JML, the assertions come from the specifications — they are checks that the specifications hold, at least for the particular execution of the program. Hence, RAC is an instrumented version of classis dynamic teseting.

8.1 Compiling classes with assertions

Compiling classes for runtime-assertion checking (RAC) is accomplished by

- compiling a program with the regular Java compiler
- compiling some (or all) of the resulting classes over with RAC enabled

The command-line to compile for RAC is the same as the command-line for Java compilation, except

- `openjml` is used instead of `javac`
- the option **--rac** is included (along with any other desired OpenJML options)

There are a few points to note:

- Both `openjml` and `javac` will compile all the classes on the command-line and any classes referred to by those classes but not yet compiled. Hence it can be useful to perform a full javac compilation first, so no unexpected files have RAC enabled.
- Assertions are compiled only into classes compiled with --**rac**, and not into library classes or super classes.
- Assertion violations are reported only for the particular execution of the program. An absence of reports does not mean that some other run of the program (with different inputs) will be assertion-violation-free.

It is helpful to understand what assertions are generated (and checked by RAC). The full set is listed here, but with more detail and examples in Appendix **??**; options described below can control which of these assertions are included. Note that preconditions and postconditions may be checked twice, once by the caller and once by the callee. At the time a given class is compiled, it does not know whether its counterpart in the caller-callee relationship will also be compiled with assertion checks; hence the precondition or postcondition is checked by both, to ensure it is at least checked once.

- well-definedness checks of any assertion or assumption, before the assertion or assumption itself is checked
- any explicit JML assert, reachable and unreachable statement
- any explicit JML assume statement (not checked by default)
- non-null checks when a object is deferenced (dot-operator or array-element operator)
- non-null checks when a reference variable or formal parameter declared NonNull is assigned
- array index is in range when an array is indexed
- checks implied by `assignable` clauses on any assignment
- checks implied by `accessible` clauses on any read in Java code
- pre-conditions and invariants of a callee, checked as assertions by the caller before calling a callee
- pre-conditions and invariants of a callee, checked as assumptions by a callee after being called but before executing the body of the callee (not checked by default)
- post-conditions and invariants of a callee, checked as assertions by a callee after executing the body of the callee
- post-conditions and invariants of a callee, checked as assumptions by a

caller after returning from a callee (not checked by default)
- More? Label with the label that is used.

8.2 Executing RAC-compiled programs

To execute a RAC-compiled program, either

- (a) run the program as usual but using `openjml-java` rather than `java`
- or (b) run the program with conventional `java` (at least V17) but include the `jmlruntime.jar` library on the classpath.

8.3 Options specific to runtime checking

8.3.1 --show-not-executable

--**show-not-executable**: (default: disabled) warns about the use of features that are not executable (and thus ignored). Some features of JML are not executable. If this option is enabled, warnings are printed during compilation when such features are used. Turning on this option can be helpful to a user unsure why a particular assertion is not being reported failing, just to be sure it is actually being compiled.

8.3.2 --show-not-implemented

--**show-not-implemented**: (default: enabled) warns about the use of features that are not yet implemented (and thus ignored). This option is on by default, but the user may wish to disable it (with –**show-not-implemented=false** in order to reduce warning messages that are not adding useful information.

8.3.3 --rac-show-source

--**rac-show-source** *choice*: (default: source; choices: none, line, source) includes source location in RAC warning messages. If this option is set to `source` then RAC assertion violation messages will include text from the source file indicating the location of the violation, in addition to the report of line number. The option can provide more helpful error information, but it also can considerably increase

the size of the compiled classes. So for large programs, it may be helpful to set this option to 'line'.

As an example, the input file

```
public class A {

  public static void main(String... args) {
    //@ assert args.length == 1;
  }
}
```

when compiled with the command

```
openjml --rac --rac-show-source A.java
```

and run with

```
openjml-java A
```

produces the output

```
A.java:4: verify: JML assertion is false
    //@ assert args.length == 1;
        ^
```

If compiled with

```
openjml --rac --rac-show-source=line A.java
```

the output is

```
A.java:4: verify: JML assertion is false
```

If compiled with

```
openjml --rac --rac-show-source=none A.java
```

the output does not even have the line numbers

```
verify: JML assertion is false
```

8.3.4 --rac-check-assumptions

--rac-check-assumptions: (default: disabled) when enabled, both assumptions and assertions are checked. Checking both gives more thorough runtime checking, but also increases the size of the RAC-enabled program considerably. If size or runtime performance becomes a problem, the user may wish to disable this feature. However, when the option is disabled, users can sometimes be confused about why an apparent violation is not reported.

This option particularly affects the checking and reporting of pre- and postconditions. When a method (the callee) is called from an another method (the caller), the preconditions of the callee are checked (an assertion) by the caller before the call, and the postconditions are assumed by the caller after the call. Within the callee, however, the preconditions are assumed at the beginning of the method execution and the postconditions are asserted at the end.

So this input file

```
1  public class A {
2
3    public static void main(String ... args) {
4      m(args.length);
5      mm(args.length);
6    }
7
8    //@ requires i == 1;
9    //@ ensures \result == 20;
10   public static int m(int i) {
11     return 10;
12   }
13
14   //@ requires i == 0;
15   //@ ensures \result == 20;
16   public static int mm(int i) {
17     return 10;
18   }
19 }
```

when compiled with the command

```
openjml --rac --rac-check-assumptions A.java
```

and run with

```
openjml-java A
```

produces the output

```
1  A.java:4: JML precondition is false
2      m(args.length);
3        ^
4  A.java:8: Associated declaration: A.java:4:
5    //@ requires i == 1;
6        ^
7  A.java:8: JML precondition is false
8    //@ requires i == 1;
9        ^
10 A.java:16: JML postcondition is false
```

```
  public static int mm(int i) {
                     ^
A.java:15: Associated declaration: A.java:16:
  //@ ensures \result == 20;
      ^
A.java:5: JML postcondition is false
    mm(args.length);
      ^
A.java:15: Associated declaration: A.java:5:
  //@ ensures \result == 20;
      ^
```

The example output shows the preconditions and postconditions each being checked twice, once by the caller and once by the callee, because both assumptions and assertions are checked at runtime.

However, if the example is compiled with

```
openjml --rac --no-rac-check-assumptions A.java
```

the output is

```
A.java:4: JML precondition is false
    m(args.length);
     ^
A.java:8: Associated declaration: A.java:4:
  //@ requires i == 1;
      ^
A.java:16: JML postcondition is false
  public static int mm(int i) {
                     ^
A.java:15: Associated declaration: A.java:16:
  //@ ensures \result == 20;
      ^
```

Here only assertions are checked: the preconditions by the caller and the postconditions by the callee.

So why not always disable this option to avoid duplication? The duplication happens because both the caller and the callee are being compiled with RAC. If, however, the callee was a library routine that was not compiled with RAC, then we would want both the postconditions and preconditions checked by the caller, and then we would want this option enabled.

8.3.5 --rac-java-checks

--rac-java-checks: (default: disabled) when enabled, runtime-assertions that check for Java language violations are enabled. Enabling this feature causes more thorough checking and causes all violations to be reported uniformly. However it also increases the size of RAC-compiled programs. If this option is disabled, RAC will not check for the violation, but Java will. For example, if there is an array index operation, JML can check that the array index is within bounds. If the JML check is disabled, Java will report a `ArrayIndexOutOfBounds` exception, so the violation will be reported to the user anyway, just through a different exception. Because of this backup Java checking and to reduce compiled code size, this option is disabled by default. However, the option is useful during testing, because then all violations of JML assertions are reported through OpenJML, so a test harness can uniformly detect and report violations during unit testing.

The discussion in §8.4 below is also important to when and how JML violations are reported.

As an example, the input file

```
public class A {

  public static void main(String ... args) {
      int i = args.length;
      int j = i/(i-i);
  }

}
```

when compiled with the command

```
openjml --rac --rac-java-checks A.java
```

and run with

```
openjml-java A
```

produces the output

```
A.java:5: JML Division by zero
      int j = i/(i-i);
                ^
Exception in thread "main" java.lang.ArithmeticException: / by
    zero
        at A.main(A.java:5)
```

The output contains first a JML error that an imminent divide-by-zero was detected. Then the program proceeds to execute the division and produces a standard Java error.

If compiled with

```
openjml --rac --no-rac-java-checks A.java
```

the output is

```
Exception in thread "main" java.lang.ArithmeticException: / by
    zero
        at A.main(A.java:5)
```

Here the JML check is omitted, so only the Java exception is reported.

8.3.6 --rac-compile-to-java-assert

--rac-compile-to-java-assert: (default: disabled) compiles RAC checks using Java asserts (which must then be enabled using `-ea`) during execution, instead of using `org.jmlspecs.utils.JmlAssertionError`. When this option is enabled, all assertion violation reporting is through Java assertion errors; that is, Option (C) in §8.4 is used despite any system properties. Furthermore, no reports will be generated at all at runtime unless the Java option `-ea` is enabled.

Need example

8.3.7 --rac-precondition-entry

--rac-precondition-entry: (default off) enables distinguishing internal Precondition errors from entry Precondition errors, appropriate for automated testing; compiles code to generate `JmlAssertionError` exceptions (rather than RAC warning messages)

TBD - should this turn on -racCheckAssumptions?

Complete the above and Need an example

8.4 Controlling how runtime assertion violations are reported

There are three ways in which a RAC-compiled program can report assertion violations. These can be controlled by properties set at the time the RAC-enabled program is *run* (not when it is *compiled*). Note that if the option **--rac-compile-to-java-assert** is enabled (§8.3.6) then option (C) below is compiled in at compile time, and the various runtime alternatives described here are no longer available.

- A) as messages printed to `System.out`. In this case the program will continue executing after printing the assertion violation and may possibly encounter and report additional violations or Java exceptions. This reporting mechanism is the default and applies if neither property `org.jmlspecs.openjml.racexceptions` nor `org.jmlspecs.openjml.racjavaassert` is defined while the program is executing. In this reporting mode, an additional useful system property is `org.jmlspecs.openjml.racshowstack`. If this property is defined, then the stack trace to an assertion violation is reported along with the violation message. This makes the output more verbose, but may make it easier to debug why a particular violation is occurring.

- B) as a thrown exception of some subtype of `org.jmlspecs.utils.JmlAssertionError`. This reporting mechanism is used if the system property `org.jmlspecs.openjml.racexceptions` is set while the program is executing. The subtype is determined by the kind of violation. Execution of the program stops with the first violation reported. Refer to list of labels

- C) as a thrown exception of the type `java.lang.AssertionError`. Execution of the program stops with the first violation reported. This is the same kind of assertion that is thrown by a Java `assert` statement. These exceptions are not thrown by default but are enabled by the Java option `-ea` or `-enableassertions`. This reporting mechanism is used if `org.jmlspecs.openjml.racjavaassert` is defined but `org.jmlspecs.openjml.racexceptions` is not. One advantage of this mechanism is that Java allows controlling assertion reporting by class and package, by customizing the `-ea` option. (See the Java documentation

for `-ea` and `-da` for specific information.)

Recall that system properties can be enabled by running the program with a command-line like

```
openjml-java -Dorg.jmlspecs.openjml.racjavaassert MyProgram
```

These examples need checking

As an example, the input file

```
1 public class A {
2
3   public static void main(String ... args) {
4       int i = args.length;
5       int j = i/(i-i);
6   }
7
8 }
```

when compiled with the command

```
openjml --rac A.java
```

and run with

```
openjml-java A
```

produces the output

```
1 A.java:5: JML Division by zero
2       int j = i/(i-i);
3                 ^
4 Exception in thread "main" java.lang.ArithmeticException: / by zero
5         at A.main(A.java:5)
```

If compiled the same way but run with

```
openjml-java -Dorg.jmlspecs.openjml.racshowstack A
```

the output is

```
1 A.java:5: JML assertion is false
2     //@ assert i == 1;
3           ^
4 org.jmlspecs.utils.JmlAssertionError: A.java:5: JML assertion is false
5     //@ assert i == 1;
6           ^
7         at org.jmlspecs.utils.Utils.createException(Utils.java:99)
8         at org.jmlspecs.utils.Utils.assertionFailureL(Utils.java:58)
9         at A.main(A.java:1)
```

If compiled the same way but run with

```
openjml-java -Dorg.jmlspecs.openjml.racexceptions A
```

the output is

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at A.main(A.java:5)
```

And if compiled the same way but run with

```
openjml-java -ea -Dorg.jmlspecs.openjml.racjavaassert A
```

the output is (Bad line numbers)

```
Exception in thread "main" java.lang.AssertionError: A.java:5: JML assertion is
    false
    //@ assert i == 1;
        ^
        at org.jmlspecs.utils.Utils.assertionFailureL(Utils.java:54)
        at A.main(A.java:1)
```

If the `-ea` option is omitted, this last example will produce no output.

Generally speaking, mechanism (A) is the easiest and most useful. However, mechanism (B) is useful for fine-grained control over which assertions are reported. Different types of violations have different *labels*, such as `Precondition` or `Invariant`. These labels are the same as the warning categories listed in Appendix B.

- If there is a system property `org.openjml.exception.`*`label`* defined for a given label, then the value of that property is expected to be the name of a class that is a subtype of `java.lang.Error`, and an exception of that class is thrown (if such an exception cannot be created, then an Error of type `org.jmlspecs.utils.JmlAssertionError` is thrown.
- If there is no such property defined, then an Error of type `org.jmlspecs.utils.JmlAssertionError$`*`label`* is thrown, if that type exists. Such a class is a nested class defined within `JmlAssertionError` and so must be part of the OpenJML runtime library. Currently only `Precondition` and `PreconditionEntry` are defined, but others may be added in the future. All such nested classes are derived from `org.jmlspecs.utils.JmlAssertionError`.
- If no such nested class is defined, then an `java.lang.Error` of type `org.jmlspecs.utils.JmlAssertionError` is thrown.

The user may include try-catch blocks to catch particular kinds of assertions. This may be useful in performing unit tests for example. A particular distinction useful in automated unit testing is between different kinds of Precondition violations. Say more here and give an example how to use – see option above

8.5 Exit code from a RAC-ed program

A program compiled with runtime assertion checks is supposed to have the same behavior as the original program except (a) it will emit assertion errors (and may halt early) and (b) it will likely have different time and space performance. In particular though, it will emit the same exit code regardless of any runtime assertion errors.

That is sometimes and sometimes not desirable. Accordingly one can set a property to determine the RAC-compiled program's exit code if assertion errors occur at runtime and the program is allowed to continue to its normal conclusion (behavior (A) in the previous section). If the program is run with the property `-Dorg.jmlspecs.openjml.racexitcode` set equal to the string representation of an integer, then that integer will be the program's exit code if any runtime assertion errors occur.

To continue the example of the previous section, the input file

```
1 public class B {
2     public static void main(String ... args) {
3         //@ assert args.length == 2;
4     }
5 }
```

when compiled with the command

```
openjml --rac B.java
```

and run with

```
openjml-java -Dorg.jmlspecs.openjml.racexitcode=42 B ; echo $?
```

produces the output

```
1 42
```

If the program is run with two arguments, as in

```
openjml-java -Dorg.jmlspecs.openjml.racexitcode=42 B 1 2; echo $?
```

then the assertion succeeds, no test is output, and the exit code is 0.

8.6 RAC FAQs

This is section describes some common problems that users encounter with OpenJML's runtime assertion checking.

8.6.1 Uncompiled fields and methods

When model or ghost fields or methods of class B are used by class A and class A is compiled with RAC, but class B is not, runtime errors will occur. This happens because the content of B.class is just what is produced by the Java compiler and does not have any JML fields or methods. No error occurs at compile time because OpenJML can see the declarations of JML fields and methods in class B; since Java compilation units (e.g., A and B separately) can be compiled separately, the system does not know until runtime that B has not been compiled with JML.

Make an example

8.6.2 Non-executable or unimplemented features

Some JML features are not executable by RAC. One example is a quantified expression over unrestricted `\bigint` or `\real` variables. Also, some JML constructs are not implemented. If the OpenJML options are set so that no warnings are issued about non-executable or not-implemented features, then some default value is used: expressions typically default to true and clauses typically default to being ignored. This can cause a difference in behavior between RAC and ESC and can also cause confusion in users when comparing RAC output to the JML specifications as written. The recommendation is to always enable the options **--show-not-implemented** and **--show-not-executable** for any crucial or final or debugging runs of OpenJML.

Make example

8.6.3 Try blocks too large

RAC adds a large amount of assertion checking into a Java method. Consequently some Java implementation limitations can be reached. One such limitation is the size of try blocks. Even methods that do not have try blocks of their own are wrapped in try blocks by RAC to check for unexpected exceptions.

A future task is to optimize RAC in a way the minimizes the extra overhead, such as by omitting runtime checks for assertions that are 'obviously' (perhaps easily statically provably) true.

Some tips to avoid this problem are these:

- Keep methods small
- Limit runtime assertions to just those needed to check crucial invariants and preconditions
- Use the --**no-rac-check-assumptions** option.

Chapter 9

OpenJML extensions to JML

The 2nd edition of the Java Modeling Language has many additions and deletions compared to JMLv1. Even so, there are language features that were intentionally omitted, primarily because those features deal with proof assistance rather than specification per se. This chapter describes language features that OpenJML provides that are not in standard JML.

The grammar of each feature is given in the same style as is used in the JML Reference Manual 2nd edition.

The --**lang** option enables a choice among JML language variants. The current options are --**lang=openjml** (the default) or --**lang=jml**. With the latter option, warnings are given for any feature that is not strict standard JML. These are only warnings, not errors, unless -**Werror** is used.

9.1 Specification statements

Specification statements are JML specifications that can be placed where a typical Java statement would be, in the body of a method or initializer block. Recall that JML specifies the behavior of methods and classes, and not the details of method implementations. Any specifications in the body of a method are there either to aid the verification attempt or to understand the relationship between specifications and implementation. Hence JML contains only a few very common specification statements. JML defines these specification statements (cf. JML Reference Manual, CH. TBD):

- `assert` statement
- `assume` statement
- block specifications
- loop specifications

Check the above list

OpenJML adds these, described in succeeding subsections:

- `check` statement (§9.1.1)
- `show` statement (§9.1.2)
- `havoc` statement (§9.1.3)
- `halt` statement (§9.1.4)
- `split` statement (§9.1.5)
- `reachable` statement (§9.1.6)

9.1.1 check statement

Grammar:

<jml-check-statement> ::=
    check *<opt-name>* *<jml-expression>* ;

Type checking requirements:

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

A `check` statement behaves just like a JML `assert` statement except for this: after a `check` statement, the predicate is *not* assumed to be true, as it is for an `assert`. Thus, in this code fragment

```
1 // c possibly null
2 //@ check c != null;
3 //@ int i = c.value;
```

a tool should give two errors: one that the `check` statement is not provable and a second that there might be a null-dereference in the `c.value` expression. In contrast, if an `assert` were used instead of the `check`, there would only be a verification failure on line 2; after that `assert`, `c != null` is presumed to be true.

A `check` statement is useful for inquiring about the truth of a given predicate

without otherwise disturbing the logic of a program. A `check` statement is considered a *soft assertion*, whereas an `assert` statement is a *hard assertion*.

9.1.2 show statement

Grammar:
<jml-show-statement> **::=** **show** *<opt-name>* *<jml-expression>* ... ;

Type information:
The expressions in the `show` statement may have any type other than `void`.

The show statement is a debugging statement and may be ignored by tools. If implemented, the expected behavior is this:

- When executed during runtime-assertion-checking, it prints out (as with `System.out.println`) the values of the given expressions. As the expressions may be JML expressions, they are not accessible to debugging of the Java program itself.
- In static checking, if a proof of a method fails with a counterexample, then the counterexample contains the values of the show-statement expressions for inspection, associated with some identifying information.

The `show` statement provides functionality similar to the `\lbl` expression, but more conveniently. As is the case for all JML expressions, the `show` statement has no side-effects.

9.1.3 havoc statement

Grammar:
<havoc-statement> **::=** **havoc** *<opt-name>* *<store-ref-expression>* **, ... :**

Grammar needed

The `havoc` statement includes a list of *<store-ref-expressions>*, just like an `assignable` clause. The effect of the `havoc` statement is that all the listed memory locations are given new values that are arbitrary except that they satisfy the type and invariant constraints for the type of the memory location. The `havoc` statement can be used to simulate an arbitrary input or the effect of a method call.

Need example

9.1.4 `halt` statement

Grammar: *<halt-statement>* `::=` **`halt`** *<opt-name>* `[ ; ]`

A `halt` statement in the body of a method causes OpenJML to stop translating statements of the method body. Only implicit or explicit assertions up to the point of the `halt` statement will be checked. This statement provides an easy way to include less or more of the body of a method in the proof attempt, in order to see where a problem with the proof may lie.

If the method body has various conditional branches or loops, the `halt` statement only stops translation for the branch in which it appears. To stop processing in all branches, a `halt` must be placed in each one. On the other hand, by placing a `halt` in some but not all branches, one can determine which branches are successfully proved and which are causing the proof to fail.

Add example

9.1.5 `split` statement

Grammar:
<split-statement> `::=` **`split`** *<opt-name>* `[` *<expression>* `];`

Type information: If the optional *<expression>* is present, it must have boolean type.

Normally OpenJML constructs a single large verification condition for a method and submits it to the back-end logic solver. The solver, which is highly optimized, finds any violations of any assertion in the verification condition. Sometimes however this VC is just too large and it needs to be broken up into smaller proof attempts.

One way to break up a proof is to use block specifications, which are part of standard JMLv2 (cf. JML Reference Manual Ch. TBD). In one proof the body of a block statement is verified against its specification and in a second proof the block specification is used as a summary to shorten the block when the rest of the method body is verified. Check that OpenJML does not need a split statement here

The `split` statement provides a second way to break up a proof. It can be used in three situations:

- Just before an `if` or `switch` statement
- Just before a loop statement (but after the loop specifications) check this
- at any statement location if the optional boolean expression is present.

Only in the last case is the optional expression permitted.

The effect of the `split` statement is to divide the monolithic proof attempt for a method into multiple proof attempts.

- If the `split` is before an `if` statement, then the proof is split in two, one for each branch of the `if`; in one proof the then branch is followed, in the other the else branch is followed.
- If the `split` is before a `switch` statement, the proof is split into multiple subproofs, one for each case of the `switch`.
- If the `split` is before a loop, then there are two subproofs, one for the body of the loop and one for the exit branch. what about do while loops
- If the `split` is a standalone statement with a boolean expression, two subproofs are constructed, one when the expression is true and one when it is false.

What about statement specs

It is permitted to have multiple `split` statements in a method body. In that case, the splits may be multiplicative, depending on where in the control flow they appear. For example, if there are two consecutive if-statements, each preceded by a `split` statement, four different verification conditions will be created. On the other hand, if an if-statement is in the then-branch of an enclosing if-statement, then there will be three proof attempts, for the then-then, then-else, and else control flows.

Each subproof is given a designator consisting of a sequence of uppercase letters. For example the four way split above would have proofs designated AA, AB, BA, BB, where the first letter indicates which branch of the first if is followed and the second letter indicates the branch of the second if. The three-way split example above would have proofs labeled AA, AB and B. These designators can be used with the **-split** command-line option.

Using a `split` command automates some manual uses of `halt` commands to select various control flow branches to test.

TODO – more on the split option

9.1.6 `reachable` statement

Grammar:
<jml-reachable-statement> ::= **reachable** *<opt-name>* [;]

The `reachable` statement asserts that there exists a feasible execution path that reaches this statement.

The examples that follow are explained by the comments:

```
1  void m1(int i) {
2     //@ assert i == 0; // ERROR: i can be any integer,
3                        //        not just 0
4  }
5  void m2(int i) {
6     if (i > 0) {
7       //@ reachable // OK - reachable in some scenario
8     }
9  }
10 //@ requires i > 0;
11 void m3(int i) {
12    if (i < 0) {
13      //@ reachable //ERROR: not reachable
14                    // with precondition and if condition
15    }
16 }
```

The `reachable` statement is especially useful for checking the feasibility of a program, answering questions such as can execution ever go down a certain execution path; it is also used to check whether the specifications for a method are accidentally contradictory, in which case the method body is not feasible. For example, verification of the following code will fail at the `reachable` statement because the precondition contradicts the else branch of the if-statement; if the precondition holds, the else branch will never be executed; *consequently the code within the else branch will not be verified either.*

```
1 //@ requires i > 0;
2 void m(int i) {
3    if (i > 0) { ... }
4    else {
5      //@ reachable
6      throw new RuntimeException("Argument not positive");
7   }
8 }
```

Reachability testing can be time-consuming, so the default verification does not check feasibilty. The reachability test is different than verification, requires a separate formulation and SMT test, and typically requires separate executions of underlying solvers, as the test is now to find at least one path that reaches the given statement. If there are multiple `reachable` statements in a method, the check is for each one of them individually; they are not required to all be reachable for the same initial state.

`reachable` statements are not useful for runtime-checking. At runtime a program can only know that it has reached a particular `reachable` statement (which is a tautology); it cannot know whether other reachable statements are reachable for other executions of a program. [1]

The `reachable` statement is subject to false positives. A `reachable` statement's success, that is, that the prover says that there is an input state that will bring about execution of the `reachable` statement, may be due to over-approximation. For example, consider

```
1  //@ ensures i > 0 ==> \result < 0;
2  public static int neg(int i) { return i > 0 ? -i : i; }
3
4  public static void m(int i) {
5    //@ assume i == 1;
6    int j = neg(i);
7    if (j == -2) {
8       //@ reachable
9    }
10 }
```

[1] A tool could check that across a whole test suite all reachable statements are in fact reached.

In checking method `m`, we use the specification of `neg`. By the given specification, a return value of `-2` from `neg` is possible, even when the value of `i` is `1`. Hence the reachable statement is deemed feasible. If a more precise specification `neg` were used, say `ensures i > 0 ==> \result == -i`, which is still partially under-specified, then a prover can tell that the reachable statement is infeasible.

Reachability analysis works with the **--check-feasibility** option. This option takes a value that states which locations in a method are to be tested for reachability. Those locations can be explicit, using the reachability statement, or implicit, such as the end of the preconditions or all return statements.

The value of the **--check-feasibility** option is a comma-separated list of the identifiers listed below, indicating the corresponding kinds of locations to check:

- **reachable** – all points in the method explicitly marked with a `reachable` statement
- **precondition** – at the beginning of the method body; checks whether there are contradictions in the preconditions and invariants
- **assert** – just before each explicit assert statement; if the execution path to the assertion is not feasible, the assertion will never be checked
- **assume** – just after each explicit assume statement; if the execution path is not feasible, there is something wrong with the predicate being assumed (or something wrong before it)
- **return** – is every return statement feasible (after computing the return value)
- **throw** – is every throw statement feasible (after computing the throw expression)
- **if** – are both branches of the if condition feasible
- **switch** – are all branches of a switch statement feasible
- **catch** – at the beginning of each catch block
- **finally** – at the beginning of each finally block
- **spec** – at the end of every statement spec block
- **call** – after any call
- **halt** – at each halt statement
- **loopcondition** – at the beginning of the loop body
- **loopexit** – on the exit branch after testing the loop condition
- **exit** - is it possible to exit the program (either normally or with an exception)

In addition there are these special identifiers:

- **none** – no checking; the default
- **basic** – includes precondition, assert, assume, reachable, exit, halt, and spec
- **all** – all the categories listed above
- **debug** – for debugging OpenJML itself

9.2 Modifiers

Write this; check against JMLRMv2

two-state, strictly_pure infer

9.2.1 skipesc and skiprac

The modifiers `skipesc` and `skiprac` are permitted on methods and classes. Their effect is to turn off any ESC (verification) or RAC compilation (respectively) for that method or for any method contained within the class (or contained in nested classes, recursively).

The same effect can be achieved using a **--method** or **--exclude** command-line option, but the modifiers allow semi-permanent disabling of, say, verification attempts of a very-long-to-verify method. Of course, for soundness, one needs to verify all methods self-consistently eventually.

9.2.2 inline

The `inline` modifier may be applied to a method that has a body. The effect is to replace a call to the method with an inlining of its body. In ESC, then, it serves to eliminate the need for a specification, as the body now serves as the statement of what the method accomplishes. This is a very basic form of specification inference and is most applicable to simple methods like getter and setter methods.

What if there is a body as well as inline? Are preconditions checked anyway? IS it enabled in RAC?

9.2.3 query and secret

Probably write a full concept section on observational purity

9.2.4 immutable

Some Java classes, such as `Integer` and `String`, create *immutable* objects: once an instance is constructed, it cannot be changed. All methods have no side-effects and there are no fields to be assigned.

This is the intent of the `immutable` modifier — to mark such kinds of classes. However sufficient questions remain so that this is still an experimental feature under discussion.

- Is the immutability shallow or deep? That is, if an immutable object captures other objects, which are then part of its representation, must those objects in turn be immutable?
- What if a method (`m(T t)`) of the immutable object calls methods of its arguments (`T.p()` which do have side effects somewhere? Then `m` itself cannot be pure.
- Must immutable classes be final? Or is immutability inherited?
- May immutable classes be derived from non-immutable parents? Then the immutable class might have mutable fields?
- The Object class might have mutable ghost fields, like `owner`. Should that prevent any Java class from being declared immutable? Even what seem like obvious candidates like `Integer`?

So at present, though `immutable` is a recognized modifier on a class, it does not imply any particular behavior or obligations.

9.2.5 @Options

The `@Options` annotation can annotate a class, interface, or method declaration. The effect is to have any command-line options present in the argument of the annotation be applied to the method or to all the methods (recursively) contained within the given class or interface declaration.

The argument of the `@Options` annotation is either a String literal or a brace-enclosed, comma-separated list of String literals, as in either `@Options("--esc-max-warnings=1")` or `@Options("-esc-max-warnings=1","-check-feasibility=bas`
For this feature, only the annotation `@Options` can be used, not a simple modifier (i.e., `options`). However, `@Options(...)` may be placed within JML annotation text so it does not affect the Java program:

```
1 //@ @Option("--esc-max-warnings=1")
2 public void m() { ... }
```

Only OpenJML (not OpenJDK) options may be applied in this way, and only those whose effect is directly on the ESC or RAC translation of the method. For example, `--specspath` is used during parsing and typechecking, and so would not be allowed to be applied to a method in this way.

9.3 Expressions

9.3.1 \exception

Just as `\result` is an expression that denotes, in an `ensures` postcondition, the value returned by a `return` statement, `\exception` denotes the exception thrown on exit from a method. Although in a `signals` clause, there already is a variable declared representing the exception, that is not true of other clauses that are evaluated in an exceptional postcondition, such as `duration` and `working_space`.

The expression `\exception`

- is null in a normal exit from the method
- has type `java.lang.Exception` except in a `signals` clause, where it has the same type as the declared variable

9.3.2 Enhancements to conditional annotations

Besides the conditional annotation syntax described in §4.5, OpenJML also allows the following.

- In expressions, the term `\key("`*key*`")`, is either a true or false Boolean literal, depending on whether the given *key* is defined or not. Quotes or an identifier?

The keys here are the same keys as are used in §4.5, defined in the same way with the **-keys** option.

9.4 Enhancements to the maps clause

In OpenJML, the `maps` clause allows a comma-separated list of storeref expressions, not just one. That is the grammar is

```
<maps-clause> ::=
     maps <storeref> ... \into <identifier> ... ;
```

9.5 Other topics to include, possibly

TODO

reasoning about captured objects (including `capture` modifier)

`non_null_elements`

adding specification types

post for old/pre declarations in specifications

\nonnullelements for collection classes

specification of lambda functions

begin end markers

inline_loop

\values

\reach

mutliple arguments for \invariant_for, \static_invariant_for

invariants method spec clause

use of for_example as feasibility

recommends-else

expanded array-range syntax; store-refs that include expressions

allow optional semicolons

functional form of \lbl

control of invariants

Chapter 10

Extending OpenJML

This chapter is barely started.

10.1 Basic Concepts

10.2 Organization of OpenJDK and OpenJML implementation

OpenJML is designed (though somewhat incompletely as yet) to be extendable without too much major surgery on the implementation. All the JML clauses, modifiers, types and the like are defined in *extension* files. These files must be compiled and combined with the build of OpenJML, perhaps as a library, but they will need to inherit from portions of the existing implementation. The process for adding new features is described in the sections of this chapter.

The user-supplied extension files do need to be found by OpenJML when it starts. These files can be placed directly in the `org.jmlspecs.openjml.ext` package and folder and compiled with the rest of OpenJML, or they can be compiled separately and linked in as part of teh calss path. If they are in a different package than `org.jmlspecs.openjml.ext`, OpenJML must be told what package they are in via the **--extensions** command-line option (which can also be defined in a properties file, cf. §4.3).

TODO- stuff to write, examples to give

10.3 Adding command-line options

10.4 Adding modifiers

10.5 Adding statement specification clauses

10.6 Adding method specification clauses

10.7 Adding class specification clauses

10.8 Adding built-in types

Chapter 11

Other OpenJML tools

11.1 Inferring specifications

This section will expanded in the future.

The ability to infer specifications, saving the work of writing them, is an anticipated addition to OpenJML.

Specifications can only be inferred accurately in limited situations. At present specifications are inferred in the situations described in the following subsections.

Need to implement and add information about how to inform user of spec inferences

11.1.1 loop_modifies clauses

The `loop_modifies` clause, if absent from a loop specification, is inferred by analyzing the pattern of assignments in the loop body. The inferred set of storerefs always includes the loop index from a `for` statement or the implied loop index for an enhanced for statement. Also the JML identifier `\count` is always included.

Other examples

11.2 Generating Documentation

This section will be added later.

11.3 Generating Specification File Skeletons

This section will be added later.

11.4 Generating Test Cases

This section will be added later.

Chapter 12

Limitations of OpenJML's implementation of JML

12.1 Soundness and Completeness

Much is made of the soundness and completeness claims of program analysis tools. In fact programs verifiers and bug finding tools use the terms *soundness* and *completeness* in different ways. One way to think about this question is in terms of the guarantees that a tool claims to make.[1] A tool can be said to be *sound* if the guarantee it makes actually holds. It is *complete* if it identifies all situations in which its guarantees do not hold. Consider the partitioning of the space of actions and results of tools shown in Fig. 12.1 from the points of view of bug-finding tools and deductive verification tools.

Bug-finders Users looking for bugs waste time analyzing bug reports that are not actual bugs; that is, they want Q_2 in Fig. 12.1 to be empty, ideally. They are

[1] Gary leavens suggested this approach to me

	P has a bug at L	P does not have a bug at L
T reports a bug at L	Q_1	Q_2
T does not report a bug at L	Q_3	Q_4

Figure 12.1: Combinations of the behavior of a program P and tool T concerning a bug at program location L

not so concerned that all bugs are reported (that is, that Q_3 is empty); rather they need to find and fix the most bugs of consequence in a fixed amount of time[24, 14, 15]. Consequently the soundness goal for a bug-finder is this: any reported bug is a true bug (Q_2 is empty). A secondary goal is completeness: all bugs are found (Q_3 is empty).

Program verifiers A program verifier, on the other hand is concerned that all bugs are reported, even if some of them, because of limitations of the tools, are not real bugs. The soundness claim for a program verifier is *all actual bugs are reported by the tool*. That is, Q_3 is empty. A secondary goal is completeness: all bugs reported are actual bugs (Q_2 is empty).

Tools cannot achieve both soundness and completeness. In practice some tradeoff between them is necessary in practical and usable tools. A bug-finder could report no bugs and be 100% sound, but also totally incomplete and thus unusable; it could report bugs everywhere and be 100% complete, but unsound and also unusable. Some researchers have advocated considering *soundiness*[22]: recognizing that tools cannot be completely sound and carefully describing in what ways they are not. Practioners are then aware of the capabilities and limitations of a tool.

In particular program verifiers typically analyze only a portion of the programming language they address. They may be sound for that portion, but they are not then sound for the whole language, unless they report a warning for any feature present that is only approximately analyzed; in that case the feature is an incompleteness. If most programs contain unimplemented features then the tool becomes much less usable, as unimplemented features may cause significant swaths of a program to be unanalyzed.

OpenJML aspires to be a program verifier for Java, so an important limitation is that it does not analyze all of Java. It does intend to warn the user of any feature in the target program that is not supported and to progressively work to implement missing features. Nevertheless we wish to be clear about what aspects of a program contribute to unsoundness or incompleteness in its goal of reporting all bugs in a program, interpreted as inconsistencies between a program and its specifications. (The question of whether a consistent combination of specification and implementation actually matches the users' intent and expectation of a program, that is, whether safety, security and correctness are actually achieved by the specification, is left to other, human, processes.)

Note at the start that all tools suffer from this potential unsoundness: tools may have bugs in them that lead to missing actual errors. And little of sophisticated program analysis tools are actually verified themselves.

12.2 Java and JML features not implemented in OpenJML — General issues

Currently OpenJML does not completely implement JML or Java. The differences are enumerated in the remainder of this chapter. Gaps in representing Java reduce soundness, as bugs in unanalyzable parts of a program are not found; gaps in implementing JML are a completeness issue as they reduce the expressiveness of the portion of JML that OpenJML can use, thereby reducing the ability to prove that a construct is correct and increasing the number of non-bugs reported.

12.2.1 Non-conservative defaults

More - particularly about binary files

12.2.2 Unchecked assumptions

JML allows the introduction of unchecked assumptions as `assume` statements and `axioms`, and it allows analyzing only a portion of a program using the `halt` statement. It is, however, straightforward to be sure that in a final verification, no such statements are present.

12.2.3 Verification of Java system libraries

To have a fully sound verification, all classes and methods used in a program must be verified. A typical program uses classes from the JDK (at least `Object`). These are not verified. Though one might hope that they are in the future, the effort to do so would be very substantial and likely require tools with capabilities more than OpenJML. Errors in the (only manually reviewed) JML specifications for the JDK are a soundness risk in verifying Java programs.

12.2.4 Java Errors

JML and OpenJML make no claims about programs that throw Java Errors, like `OutOfMemoryError`, whether they are caught and handled internally or whether they cause a program abort. For example, a program might be able to be specified and verified that it never crashes with an `Exception`, but the same cannot be said for an `Error`.

12.2.5 Non-sequential Java

JML makes no claims to specify non-sequential Java. Likely, JML needs additional capabilities to do so effectively. There are some language features that are the start of such support: `monitored_for`, `monitored`, and operations on sets of locks (`\lockset` and `\max`).

12.2.6 Reflection

JML does not provide language features to specify or reason about reflection.

12.2.7 Class loading

JML does not provide language features to specify or reason about class loading.

12.2.8 Modules and annotation processing

OpenJML does not implement anything special for either Java modules or Java's annotation processing. Nor does JML define any behavior regarding these Java features.

12.3 Java and JML features not implemented in OpenJML — Detailed items

Also discuss – static initialization,

12.3.1 Clauses and expressions

These JML features are parsed and typechecked but not otherwise implemented in either ESC or RAC.

- `\only_assigned`
- `\only_accessed`
- `\only_captured`
- `\only_called`
- `\only_assigned`
- `duration`
- `working_space`
- `\duration`
- `\working_space`
- `\space`

12.3.2 Termination

OpenJML does prove termination of loops, but it does not yet prove termination of recursive or mutually recursive calls. This requires working out the usability and semantics of the `measured_by` clause and default well-founded measures for termination.

12.3.3 Redundancy

OpenJML does not fully implement the redundancy features of JML. OpenJML currently

- treats the redundant keywords precisely like their non-redundant counterparts and
- ignores the `implies_that` and `for_example` specification cases.

12.3.4 Arithmetic mode

- OpenJML does not implement `code_bigint_math`
- OpenJML does not consistently implement floating point mathematics
- OpenJML does not interpret the `strictfp` Java modifier, though when floating point is supported, it will only be for `strictfp` (that is IEEE 754

compliant) calculations.

12.3.5 Quantifiers

OpenJML does not support the `\sum`. `\product`, and `\num_of` quantifiers, nor the set comprehension expression.

12.3.6 Static initialization

Verification of reentrant static initialization and the `uninitialized` keyword is not yet completed.

12.3.7 model import statement

OpenJML currently translates a JML model import statement into a regular Java import statement. Consequently, names introduced in a model import statement are visible in both Java code and JML annotations. This has consequences in the situation in which a name is imported both through a Java import and a JML model import. Consider the following examples of involving packages `a` and `b`, each containing a class named `X`.

In these two examples,

```
1 import a.X;
2 //@ model import b.X;
```

```
1 import a.*;
2 //@ model import b.*;
```

the class named `X` is imported by both an import statement and a model import statement. In JML, the use of `X` in Java code unambiguously refers to `a.X`; the use of `X` in JML annotations is ambiguous. However, in current OpenJML, the use of `X` in both contexts will be identified as ambiguous.

In

```
1 import a.*;
2 //@ model import b.X;
```

a use of `X` in Java code refers to `a.X` and a use in JML annotations refers to `b.X`. However, in current OpenJML, both uses will mean `b.X`.

However,

```
1 import a.X;
2 //@ model import b.*;
```

is unproblematic. Both JML and OpenJML will interpret `X` as `a.X` in both Java code and JML annotations.

12.3.8 Model programs

OpenJML only partially implements model programs, which includes these features of JML:

- the `extract` modifier and clause
- the `choose` clause
- the `choose_if` clause
- the `or` clause
- the `returns` clause
- the `breaks` clause
- the `continues` clause

12.3.9 Universe types

OpenJML does not implement JML's Universe types, including `peer`, `rep`, `readonly`, `\peer`, `\rep`, `\readonly`.

Chapter 13

Contributing to OpenJML

Up to date information for OpenJML developers is found on the OpenJML GitHub wiki, at https://github.com/OpenJML/OpenJML. Here we give an outline of the relevant topics, but do not describe them in detail, so as not to repeat information which is more easily kept up to date on line.

The source programming language for OpenJML is Java. OpenJML builds on the OpenJDK reference java compiler (https://openjdk.java.net).

13.1 GitHub

The GitHub project named OpenJML (github.org/OpenJML) holds a number of related repositories (some of them no longer maintained):

- `OpenJML`: contains the core software for OpenJML, including the modified OpenJDK and the tests. The relevant top-level directories in this repo are
  - `OpenJDKModule`
  - `OpenJMLTest`
  - The other top-level folders are no longer used
- `JMLAnnotations`: the source for the `org.jmlspecs.annotation` package
- `Specs`: the source for the JML specifications for the Java system library classes
- `Solvers`: binary instances of SMT solvers that are released with OpenJML.

- `openjml.github.io`: the repository holding the material for the OpenJML website at www.openjml.org, including the tutorial material.

Other important materials that should be maintained and improved:

- A wiki describing how to create and use a development environment for OpenJML (https://github.com/OpenJML/OpenJML/wiki)
- The issue reporting tool for recording and commenting on bugs or desired features (https://github.com/OpenJML/OpenJML/issues)
- The https://github.com/JavaModelingLanguage/RefMan repository, which contains discussions of the definition and semantics of JML, is more closely related to JML itself, but is very relevant to the ongoing development of OpenJML.

These repositories are out of date (and may be deleted from the HEAD of the repository):

- `OpenJMLDemo`: demo material for OpenJML
- `OpenJML-UpdateSite`: the update site for the Eclipse plug-in
- `SMTSolvers`: an Eclipse feature plug-in containing the Solvers project, so the solvers can be distributed through an update site
- `jdk8u-dev-langtools`: an obsolete snapshot of the OpenJDK8 sources
- try-openjml
- `OpenJMLFeature`
- openjml-installer

13.2 User documentation

User-facing documentation consists of the following:

- The github-pages website accessible at www.openjml.org, which includes descriptive information (e.g., how to install) and a tutorial. The sources for this set of web pages are in the `openjml.github.io` repo listed above.
- This document, the *OpenJML User's Guide*. This is a LaTeX document maintained in OverLeaf, with pdfs distributed with OpenJML releases. You may need an invitation to have access to the LaTeX source material. (https://www.overleaf.com/project/620c2512d552cc226f5f4c94)
- The JML Reference Manual is an endeavor independent of but closely re-

lated to tool projects like OpenJML. It is maintained in Overleaf at https://www.overleaf.com/project/5ceee26404c2854a1590029f

The domain name www.openjml.org is currently maintained at NameCheap.

13.3 Maintaining the development wiki

The development wiki at https://github.com/OpenJML/OpenJML/wiki is a native GitHub wiki. Its intention is to record the processes and policies followed in OpenJML development. Changes to the infrastructure should be recorded here, sufficient to allow new developers to create a correct development environment, run tests, create releases on GitHub, etc.

13.4 Issues

Bugs, new feature requests, user problems and the like are recorded in the GitHub Issues tool for the project (https://www.github.com/OpenJML/OpenJML/issues). The issues list is somewhat polluted by issues imported from the old Sourceforge site, but those that do not concern OpenJML are all more than a decade old and have been closed on that account. This list is the record of questions, bugs and of some of the feature requests.

OpenJML does not use the project management features of GitHub.

13.5 Creating and using a development environment

13.5.1 Setup

The instructions for creating a development environment are on the wiki at https://github.com/OpenJML/OpenJML/wiki/OpenJML-Development-Environment-Setup. The process is to clone several GitHub repos in sibling folders.

13.5.2 Building OpenJML

The build instructions are at https://github.com/OpenJML/OpenJML/wiki/Building-OpenJML.

The build is Makefile-driven, using modest additions to the OpenJDK Makefile. The Makefile is in the `OpenJDKModule` folder:

- `make openjml` builds the executables
- `make release` builds a trial release
- `make release-test` runs a smoke test on the most recent trial release build

There are three relevant files produced and placed in `OpenJDKModule`:

- `openjml` – the OpenJML tool (an enhancement of `javac`)
- `openjml-java` – an enhancement of `java`, which includes the runtime libraries needed for running RAC-compiled executables
- `jmlruntime.jar` – the runtime library needed to execute RAC-compiled files with standard `java` (not needed when running with `openjml-java`)

13.6 Running tests

The tests are organized as unit and functional tests in the `OpenJMLTest` folder. The Makefile at the top-level of that folder has these relevant targets:

- `make openjml` – builds the executables
- `make openjml-test` – runs all the tests, which takes about TBD minutes. Thus it is often more convenient to run different sections of tests separately or in parallel.

Say more about tests

13.7 Deploying a release

Releases of OpenJML are built and deployed through GitHub, using GitHub actions. The description of the release process is maintained here: https://github.com/OpenJML/OpenJML/wiki/CreatingReleases.

The `OpenJML` repo has a `master-module` branch. Development work is performed on the `development-module` branch. Once a release candidate is ready, it is merged into the local `master-module` branch and tested there. When

ready, it is pushed to GitHub. The `git push` automatically initiates a workflow action that then builds OpenJML for Mac and Ubuntu platforms, performs quick tests on those releases, constructs release zip files and creates a public release on GitHub.

The human in charge of the release need only verify that the release built and deployed successfully, edit the release notes, and then "publish" the release on GitHub.

13.8 Updating to newer versions of OpenJDK

As newer versions of Java are defined and corresponding releases of OpenJDK are available, one needs to merge the changes in OpenJDK into the OpenJML source. The process is a bit complex and can involve significant manual labor and debugging. It essentially consists of these steps.

- Within the OpenJML repo, on the `openjdk-src` branch, copy the source code for the new OpenJDK such that the HEAD of the branch is a faithful copy (with all additions, modifications, and deletions) of the new version of the OpenJDK source. A diff between a copy of OpenJDK and the HEAD of the branch should produce no consequential differences.
- On a new working branch off of a fully working version of OpenJML on the `development-module` branch, merge `openjdk-src` into that new branch.
- In practice (in the past), considerable review of the changes produced by that merge, as well as resolution of many merge conflicts, is needed to generate a new working version of OpenJML consistent with the new version of OpenJDK.
- When OpenJML is again working satisfactorily, merge it back to `development-module` and to `master-module` and generate a new release of the updated version of OpenJML.

Appendix A

Command-line options

These tables reproduce for convenience Tables 5.1 and 5.2 in the body of the text.

Options inherited from OpenJDK See the Java documentation for more detail	
@<*filename*>	read options from a file. *This is implemented only for Java options, not* OpenJML *options*
-Akey	options to pass to annotation processors
--add-modules <*modulelist*>	[§5.10] see Java documentation re modules
-bootclasspath <*path*> --boot-class-path <*path*>	See Java documentation
-cp <*path*> -classpath <*path*> --classpath <*path*>	[§4.2] location of input class files
-d <*directory*>	location of output class files
-deprecation	warn about use of deprecated features
--enable-preview	enables preview language features
-encoding <*encoding*>	character encoding used by source files
-endorsedirs <*dirs*>	see Java documentation
-extdirs <*dirs*>	see Java documentation
-g	generate debugging information
-h <*directory*>	location of generated header files
-? -help --help	[§5.5] output (Java and JML) help information
--help-extra	[§5.5] help about extra options

Options inherited from OpenJDK (cont.) See the Java documentation for more detail	
-implicit	whether or not to generate class files for implicitly referenced classes
-J<*flag*>	flags for the runtime system
--limit-modules <*modulelist*>	[§5.10] see Java documentation re modules
-m <*module*> --module <*module*>	[§5.10] see Java documentation re modules
--module-path <*path*>	[§5.10] see Java documentation re modules
--module-source-path <*path*>	[§5.10] see Java documentation re modules
--module-version <*version*>	[§5.10] see Java documentation re modules
-nowarn	[§5.5] show only errors, no warnings
-p <*path*>	[§5.10] like --module-path see Java documentation re modules
-parameters	see Java documentation
-proc	see Java documentation re annotation processing
-processor <*classes*>	see Java documentation re annotation processing
--processor-module-path <*path*>	see Java documentation re annotation processing
-processorpath <*path*> --processor-path <*path*>	where to find annotation processors
-profile	see Java documentation
--release <*release*>	target release for compilation
-s <*directory*>	location of output source files
-source <*release*> --source <*release*>	the Java version of source files
-sourcepath <*path*> --source-path <*path*>	[§4.2] location of source files
--system <*jdk*>	see Java documentation
-target <*release*> --target <*release*>	the Java version of the output class files
--upgrade-module-path <*path*>	[§5.10] see Java documentation re modules
-verbose	[§5.5] verbose output for Java compiler only, not OpenJML
-version --version	[§5.5] output (OpenJML) version
-X	[§5.5]Java non-standard extensions
-Werror	[§3.4]treat warnings as errors

Table A.1: OpenJML options inherited from Java. See the text for more detail on each option.

Options specific to JML Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
--arithmetic-failure *<mode>*	[§4.10.1] sets the mode for arithmetic checks: hard, soft (the default) or quiet
[-]-check	[§5.3] typecheck only (**--command=check**)
--check-accessible -checkAccessible	[§7.3.4.1] whether to check accessible clauses (default: true)
[-]-check-feasibility *<list>* -checkFeasibility *<list>*	[§7.2] kinds of feasibility to check
[-]-check-specs-path -checkSpecsPath	[§4.2] warn about non-existent specs path entries
[-]-code-math *<mode>*	[§4.10] arithmetic mode for Java code (default: safe)
[-]-command *<action>*	[§5.3] which action to do: `check esc rac compile`, default is `check`
[-]-compile	[§5.3] typecheck JML but compile just the Java code (**--command=compile**)
[-]-counterexample -ce	[§7.3.5] show a counterexample for failed static checks
[-]-defaults *<list>*	enables various default behaviors TBD
[-]-determinism	EXPERIMENTAL: ???
--dir *<dir>*	[§5.4] argument is a folder or file; enables processing all .java files in a folder
--dirs	[§5.4] subsequent arguments are folders or files (until an argument is an option)
[-]-esc	[§5.3] do static checking (**--command=esc**)
--esc-bv -escBV	[§4.11] whether to use bit-vector arithmetic (default: auto)
--esc-max-warnings *<n>* -escMaxWarnings	[§7.3.5] max number of verification errors to report in -esc
-escMaxWarningsPath	TBD? KEEP THIS?
[-]-exec *<file>*	[§7.3.2] file path to prover executable
[-]-exclude *<patterns>*	[§7.3.3] paths to exclude from verification
[-]-extensions *<classes>*	[§10] comma-separated list of extensions classes and packages
[-]-inline-function-literal	EXPERIMENTAL ?
-java	[§5.3] use the native OpenJDK tool
-jml	[§5.3] process JML constructs
-jmldebug	[§5.5] very verbose output (includes -progress) (**--verboseness=4**)
[-]-jmltesting	changes some behavior for testing (default: false)
[-]-jmlverbose	[§5.5] JML-specific verbose output (**--verboseness=3**)
[-]-keys	[§4.5] define keys for optional annotations
[-]-lang *<language>*	[§9] the JML variant to use
[-]-logic *<name>*	[§7.3.2] name of SMT logic to use (default: ALL)

Options specific to JML (cont.) Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
[-]-method <*patterns*>	[§7.3.3] methods to include in verification
--nonnull-by-default -nonnullByDefault	[§5.4] values are not null by default
[-]-normal	[§5.5] only outputs errors; no pother progress information (**--verboseness=1**)
--nullable-by-default -nullableByDefault	[§5.4] values may be null by default
[-]-osname <*name*>	[§7.3.2] Operating System name to use in selecting prover (default: "" (auto), or one of `macos`, `linux`, `windows`)
[-]-progress	[§5.5] outputs errors, warnings, progress and summary information (**--verboseness=2**)
[-]-properties <*file*>	[§4.3] property file to read (value required)
[-]-prover <*name*>	[§7.3.2] prover to use (default: z3-4.3)
-purityCheck	[§5.4] check for purity
[-]-quiet	[§5.5] no informational output (**--verboseness=0**)
[-]-rac	[§5.3] compile runtime assertion checks (**--command=rac**)
--rac-check-assumptions -racCheckAssumptions	[§8.3.4] enables (default on) checking assume statements as if they were asserts
--rac-compile-to-java-assert -racCompileToJavaAssert	[§8.3.6] compile RAC checks using Java asserts
--rac-java-checks -racJavaChecks	[§8.3.5] enables (default on) performing JML checking of violated Java features
-racMissingModelFieldRepSource	TBD
-racMissingModelFieldRepBinary	TBD
--rac-precondition-entry -racPreconditionEntry	TBD
--rac-show-source -racShowSource	[§8.3.3] includes source location in RAC assertion failure messages
[-]-require-white-space	[§6.1] whether white space is required after an @ (default: false)
[-]-show	[§5.5] prints the details of source transformation (default: false)
--show-not-executable -showNotExecutable	[§8.3.1] warn about features not executable, in --rac operations (default: TBD)
--show-not-implemented -showNotImplemented	[§5.4] warn about features not implemented (default: TBD)
--silent	[§5.5] turns off all (error, warning, informational) output except the error code (**--verboseness=-1**)
--show-skipped -skipped	[§7.3.3] show methods whose proofs are skipped (default: true)

Options specific to JML (cont.) Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
--smt *filename*	[§7.3.9] where to write generated SMT files (for off-line use or inspection)
[-]-solver-seed	[§7.3.9] seed to pass on to the SMT solver (default: 0 - no seed)
[-]-spec-math *<mode>*	[§4.10] arithmetic mode for specifications (default: bigint)
--specs-path -specspath	[§4.2] location of specs files
[-]-split	[§9.1.5] splits proof of method into sections
--stop-if-parse-errors -stopIfParseErrors	[§6.1] stop if there are any parse errors (don't do type checking or verification attempts)
-staticInitWarning	TBD
[-]-subexpressions	[§7.3.5] show subexpression detail for failed static checks (default: false)
[-]-timeout *<seconds*	[§7.3.9] timeout for individual prover attempts (default: TBD)
[-]-trace	[§7.3.5] show a trace for failed static checks (default: false)
[-]-triggers	enable SMT triggers (default: true)
-typeQuants	TBD
[-]-verboseness *<n>*	[§5.5] level of verboseness (0=quiet .. 4=jmldebug) (default: 1, -normal)
[-]-verify-exit *<n>*	[§7.3.9] exit code for verification errors (default: 6)
[-]-warn *<list>*	[§5.8] comma-separated list of warning keys (default: no keys)

Table A.2: OpenJML options. See the text for more detail on each option.

Appendix B

Static and Runtime verification failure examples

This Appendix lists, in tables below, the various kinds of verification failures that OpenJML detects. Subsequent subsections provide examples of the most common of these. The table entries contain links to the appropriate example subsection.

To simplify language, the descriptions of failures may say that a failure is issued when a particular condition is false. In RAC this is the case: the assertion is found to be false in the particular execution of the program. For ESC, it is more accurate to say that OpenJML could not establish that the condition is always true; there may be a counterexample, but it may also be that the necessary proof is too complex for the prover.

B.1 Tables

The various warnings issued by ESC or RAC are grouped into categories to make them easier to understand.

- Assertions or verification conditions generated by the semantics of Java and JML are reported by either ESC or RAC. These are listed in Table B.1
- Assumptions generated by the semantics of Java and JML are just assumed and not validated by ESC; RAC can optionally check them, under control of the option **--rac-check-assumptions** (§8.3.4). These are listed in Table B.2.
- Some items are similarly named, beginning with either `Possibly`... or `Undefined`... . The `Possibly` label is used if the condition cannot be ruled out at the given location in Java code; the `Undefined`... label is used where the condition makes a JML expression not well-defined.

Table B.1: Static warnings about assertions. These warnings are reported in RAC if the given condition is found to be false when executing the program; the warnings are reported in ESC if the prover cannot prove the condition is always true.

Warning class	Description
`Accessible`	an expression uses memory locations that violate an `accessible` clause
`ArithmeticCastRange`	[§B.3] the argument for an arithmetic cast operation is out of range for the target type
`ArithmeticOperationRange`	[§B.4] the result of an arithmetic operation is out of range for its result type
`Assert`	[§B.5] an explicit assert cannot be proved valid
`AssumeCheck`	TBD - assumption?
`Assignable`	[§B.6] an assignment or method call violates an `assignable` clause
`Axiom`	TBD - assumption
`Callable`	a method call violates a `callable` clause
`Constraint`	[§B.8] a `constraint` clause is not proved valid as part of a method postcondition
`ExceptionalPostcondition`	[§B.9] an exceptional postcondition (`signals` clause) is not proved valid

Static warnings about assertions (cont.)	
Warning class	Description
`ExceptionList`	an exception is thrown that is not in the `signals_only` exception list
`Initially`	an `initially` clause is not valid as part of a constructor postcondition
`Invariant`	
`InvariantReenterCaller`	
`InvariantEntrance`	
`InvariantExit`	
`InvariantExceptionExit`	
`InvariantExitCaller`	
`LoopCondition`	
`LoopDecreases`	the value in a loop `decreases` clause does not decrease in a loop iteration
`LoopDecreasesNonNegative`	the value in a loop `decreases` clause is negative at the beginning of a loop iteration
`LoopInvariant`	a loop invariant is not valid after the body of a loop
`LoopInvariantAfterLoop`	a loop invariant is not valid on exit from the loop
`LoopInvariantBeforeLoop`	a loop invariant is not valid before the first iteration of the loop
`NullCheck`	
`NullField`	as part of the postcondition of a method, a class field declared non_null cannot be proved to be not null
`PossiblyBadCast`	a reference expression cannot be proved to have the type requested in the cast
`PossiblyBadArrayAssignment`	assignment of a reference to an array where the reference type is not a subtype of the underlying array index type (a Java `ArrayStoreException`)
`PossiblyNegativeIndex`	[§B.11] the index of an array index operation is negative
`PossiblyNegativeSize`	[§B.12] an array creation expression has a negative size
`PossiblyNullDeReference`	an expression being dereferenced is null
`PossiblyNullField`	a NonNull field has a null value when checked as part of invariants CHECK
`PossiblyNullValue`	the value for a switch, throw, or synchronized statement is null
`PossiblyNullUnbox`	a null reference is being unboxed to a primitive
`PossiblyNullAssignment`	a null value is being assigned to a NonNull location
`PossiblyNullInitialization`	a NonNull field or variable is being initialized with a null value

Static warnings about assertions (cont.)	
Warning class	Description
`PossiblyTooLargeIndex`	[§B.13] the index of an array index operation is larger or equal to the array length
`PossiblyDivideByZero`	the denominator of a division operation is 0
`PossiblyLargeShift`	the shift amount in a left shift operation is larger or equal to the number of bits in the left-hand argument (this is not illegal in Java, but usually surprises users)
`Postcondition`	[§B.14] a postcondition (`ensures` clause) is not valid
`Precondition`	[§B.15] the composite precondition of a method being called cannot be proved valid
`Reachable`	there is no execution path to a `Reachable` statement (ESC only)
`Readable-if`	a field is read when the readable-if condition is not valid
`StaticInit`	invariants or non-nullness of fields cannot be proved valid in static initialization
`UndefinedBadCast`	within a JML expression, a reference expression cannot be proved to have the type requested in the cast
`UndefinedDivideByZero`	the denominator of a division operation is 0 in a JML expression
`UndefinedNegativeIndex`	the index of an array index operation is negative in a JML expression
`UndefinedNegativeSize`	the size of an array is negative in a JML expression
`UndefinedNullDeReference`	an expression being dereferenced is null in a JML expression
`UndefinedNullUnbox`	a null reference is being unboxed to a primitive in a JML expression
`UndefinedNullValue`	in a JML expression, an expression in a switch, throw or syncronized expression is null
`UndefinedPrecondition`	the precondition of a (pure) method being called in a JML expression does not hold
`UndefinedTooLargeIndex`	the index of an array index operation is larger or equal to the array length in a JML expression
`Unreachable`	there is an execution path to a `unreachable` statement
`Writable-if`	a field is written when the readable-if condition is not valid

Table B.2: RAC warnings about assumptions (RAC only). These warnings are enabled only when **-rac-check-assumptions** is enabled.

Warning class	Description
`ArrayInit`	
`ArgumentValue`	
`Assignment`	
`Assume`	an explicit assume statement is found to be invalid
`CatchCondition`	
`ImplicitAssume`	reported when an implicit assumption, generated internally by OpenJML, is found to be invalid
`LoopInvariantAssumption`	
`Lbl`	
`MethodAxiom`	
`MethodDefinition`	
`NullField`	a class field designated non_null is found to be null when read
`Precondition`	reported when the composite precondition of a method called within the body of the method being checked is found to be invalid during execution (check occurs in callee)
`ReceiverValue`	
`Return`	
`SwitchValue`	
`Synthetic`	
`Termination`	

B.2 Examples

For convenience, the failures are listed in alphabetical order by warning id, as given in an error message.

Each failure is illustrated with an example. In each case the example is a class `Demo.java`. To run these examples, prefix the `openjml` and `\openjml-java` executables with the path to the installation folder, or put the installation folder on your `$PATH`.

The results of running RAC on each example are similar and not shown. To run RAC, include in the `Demo` class this `main` method:

```
1   @org.jmlspecs.annotation.SkipEsc
2   @org.jmlspecs.annotation.SkipRac
3   public static void main(String ... args) {
4      int i = args.length == 0 ? 0 : Integer.parseInt(args[0]);
5      demo(i);
6   }
```

Then compile the `Demo class` with the command

```
openjml --rac Demo.java
```

and run it with the command

```
openjml-java -cp . Demo
```

Adding different numeric arguments to the end of the command will elicit different behaviors.

List the common ones? Brings the appendix here?

B.3 `ArithmeticCastRange` warning

The `ArithmeticCastRange` failure message is issued whenever an explicit cast operation might cause a truncation in the value.

```
1  public class Demo {
2
3    //@ requires i >= 0 && i < 32768;
4    static public void demo(int i) {
5      short k = (short)i;
6      byte b = (byte)i;
```

```
7     }
8  }
```

The result of ESC is

```
1  Demo.java:6: warning: The prover cannot establish an assertion (
       ArithmeticCastRange) in method demo
2       byte b = (byte)i;
3                ^
4  1 warning
```

Here the precondition limits the value of the argument `i` to be within the range of the `short` data type. So no message is issued for the cast to a `short`. However the same is not true of the cast to `byte`, so OpenJML warns about this cast.

The semantics of Java permit casts to truncate the integer values in this way, so the program is not in error. However, it may not be what the writer intended. If the intention is indeed to truncate the value, then the warning can be safely ignored.

B.4 `ArithmeticOperationRange` warning

The `ArithmeticOperationRange` verification failure is issued whenever an arithmetic operation cannot be assured to not cause an over or underflow. Note that over or underflow is a property of the operation, not of any subsequent assignment of the intermediate value produced by the operation.

```
1  public class Demo {
2
3    static public void demo(int i) {
4      int kkk = i + i + i;
5      int k = i * i;
6    }
7
8    //@ requires i >= 0 && i < 32000;
9    static public void demo2(int i) {
10     int kk = i + i;
11     int k = i * i * i * i;
12   }
13 }
```

The result of ESC is

```
1  Demo.java:5: verify: The prover cannot establish an assertion (
       ArithmeticOperationRange) in method demo: int multiply
       overflow
2       int k = i * i;
3                 ^
4  Demo.java:4: verify: The prover cannot establish an assertion (
       ArithmeticOperationRange) in method demo: underflow in int
       sum
5       int kkk = i + i + i;
6                   ^
7  Demo.java:4: verify: The prover cannot establish an assertion (
       ArithmeticOperationRange) in method demo: overflow in int sum
8       int kkk = i + i + i;
9                   ^
10 Demo.java:4: verify: The prover cannot establish an assertion (
       ArithmeticOperationRange) in method demo: overflow in int sum
11      int kkk = i + i + i;
12                      ^
13 Demo.java:4: verify: The prover cannot establish an assertion (
       ArithmeticOperationRange) in method demo: underflow in int
       sum
14      int kkk = i + i + i;
15                      ^
16 Demo.java:11: verify: The prover cannot establish an assertion (
       ArithmeticOperationRange) in method demo2: int multiply
       overflow
17      int k = i * i * i * i;
18                        ^
19 Demo.java:11: verify: The prover cannot establish an assertion (
       ArithmeticOperationRange) in method demo2: int multiply
       overflow
20      int k = i * i * i * i;
21                    ^
22 7 verification failures
```

In method `demo`, the value of the argument is unconstrained, so it is possible that an overflow or underflow can occur on addition or multiplication. In method `demo2`, the value is constrained, so addition overflow and underflow cannot occur.

The semantics of Java permits integer operations to overflow and wrap-around in 2's-complement arithmetic. So if intended, the operation is not illegal; how-

ever it can cause confusion. For instance, in Java, $(x+1) > (y+1)$ does not mean $x > y$, because y might be the maximum value of an `int`, and $y+1$ the minimum value.

Even if an operation's result is out of range, the result is still the result Java would give and no assumptions are made that restrict the operands' values.

B.5 `Assert` warning

The `Assert` failure is issued whenever an explicit JML assert statement is false.

```
1  public class Demo {
2
3    static public int demo(int i) {
4      if (i > 0) return 1;
5      //@ assert i < 0;
6      return i;
7    }
8
9    //@ requires i >= 0;
10   static public int demo2(int i) {
11     if (i > 0) return 1;
12     //@ assert i == 0;
13     return i;
14   }
15 }
```

The result of ESC is

```
1 Demo.java:5: warning: The prover cannot establish an assertion (
    Assert) in method demo
2     //@ assert i < 0;
3         ^
4 1 warning
```

Note that the assert in method `demo2` does not provoke a verification failure message because the combination of the precondition for the method and the branch condition on the line above imply that the assert is valid.

B.6 `Assignable` warning

TBD - write this

B.7 `Assume` warning (RAC only)

`assume` statements are a means to state conditions that are known to be true, but might not be provable by OpenJML; they may also be used to restrict the range of expected values for some quantities at a given point in the program. ESC assumes the predicate is true and uses it to establish later conditions.

RAC has the option to check if indeed the predicate in an `assume` statement is true, by using the **--rac-check-assumptions** option.

Thus this code

```
1  public class Demo {
2
3    static public int demo(int i) {
4      if (i > 0) return 1;
5      //@ assume i < 0;
6      return i;
7    }
8    @org.jmlspecs.annotation.SkipEsc
9    @org.jmlspecs.annotation.SkipRac
10   public static void main(String ... args) {
11      int i = args.length == 0 ? 0 : Integer.parseInt(args[0]);
12      demo(i);
13   }
14 }
```

compiled with

```
openjml --rac --rac-check-assumptions Demo.java
```

and run with

```
openjml-java -cp . Demo 0
```

results in

```
1 Demo.java:5: JML assumption is false
2     //@ assume i < 0;
3         ^
```

Without the **--rac-check-assumptions** option, no output is emitted.

B.8 `Constraint` warning

A `Constraint` failure is issued when the property stated in a `constraint` clause cannot be assured to hold at the exit of a non-constructor method. The `constraint` clause is shorthand for a postcondition that would be part of each behavior of each method's specification. A `constraint` is typically used to state relationships between pre- and post-states that should be maintained by each method.

The following example shows a case where the constraint states that the `count` value will increase in each method:

```
1  public class Demo {
2
3    private /*@ spec_public */ int count;
4
5    //@ public constraint count > \old(count);
6
7    //@ assignable count;
8    public void increment() {
9      count++;
10   }
11
12   //@ assignable \nothing;
13   //@ ensures \result == count;
14   public int count() {
15     return count;
16   }
17 }
```

That property is true for the `increment()` method, but it is not true for the `count()` method. If the writer intended that `count` record the number of method calls made, then `count()` should also increment the `count` field. On the other hand, if `count` is just the number of `increment()` calls, then the constraint should use $>=$ instead of $>$. The specification and implementation are inconsistent, but without knowing more, we cannot say which is incorrect. In any case, OpenJML issues a warning:

```
1 Demo.java:15: warning: The prover cannot establish an assertion (
    Constraint: Demo.java:5: ) in method count
2     return count;
3     ^
4 Demo.java:5: warning: Associated declaration: Demo.java:15:
```

```
5   //@ public constraint count > \old(count);
6               ^
7 2 warnings
```

B.9 `ExceptionalPostcondition` warning

The `ExceptionalPostcondition` warning is issued when the exceptional postcondition, that is, the `signals` clause, of some behavior of the method cannot be proved true. The exceptional postcondition is the conjunction, in order, of the *signals* clauses of the behavior; note that the implicit postcondition of a `signals` clause is, if the method terminates with an exception and the exception's type is an instance of the named exception (including any subclass of the exception), then the stated condition must be true. That is, for each clause of the form

`signals` (*Exc e*)*expr*;

for an exception type (subclass of `java.lang.Exception` *Exc* and arbitrary variable *e*, the condition

(*e* `instanceof` *Exc*) $\rightarrow$ *expr*

must be true, if the method terminates with an exception.

Remember that JML makes no assurances of behavior if a method terminates with a `java.lang.Throwable` that is not a `java.lang.Exception`. Also all clauses of a behavior apply only in cases in which the precondition of the behavior is true.

In the following example of an `ExceptionalPostcondition` warning, the specification of `demo` says that on exit from the method the value of `field` will be set to the value of the argument `i`, whether the method exits normally or exceptionally. We can see by inspection that the method `init` does nothing. However, the specification of `init`, which is all that is used in checking the behavior of `demo`, says nothing about its behavior. In particular, according to `init`'s specification, `init` may throw a runtime exception; if it does then the assignment to `field` in method `demo` is skipped and the `signals` clause does not hold.

```
1 public class Demo {
2
3   static public int field;
4
5   //@ ensures  field == i;
6   //@ signals (Exception e) field == i;
```

```
7   static public void demo(int i) {
8     init();
9     field = i;
10  }
11
12  static void init() {
13  }
14 }
```

Applying ESC to this example indeed produces an `ExceptionalPostcondition` warning:

```
1 Demo.java:8: warning: The prover cannot establish an assertion (
    ExceptionalPostcondition: Demo.java:6: ) in method demo
2     init();
3         ^
4 Demo.java:6: warning: Associated declaration: Demo.java:8:
5   //@ signals (Exception e) field == i;
6       ^
7 2 warnings
```

B.10 `Initially` warning

An `Initially` failure message is issued when the property stated in an `initially` clause cannot be assured to hold at the exit of a constructor. The `initially` clause is shorthand for a postcondition that would be part of each behavior of each constructor's specification, including any unwritten default specification any unwritten default constructor.

The following example illustrates the combination of an `initially` clause and a default constructor:

```
1 public class Demo {
2
3   public int count;
4
5   //@ public initially count > 0;
6
7 }
```

The result of ESC on this example is

```
Demo.java:1: warning: The prover cannot establish an assertion (
    Initially: Demo.java:5: ) in method Demo
public class Demo {
       ^
Demo.java:5: warning: Associated declaration: Demo.java:1:
  //@ public initially count > 0;
            ^
2 warnings
```

Here the default constructor leaves the field `i` at its default value of 0, in violation of the `initially` clause. Hence, OpenJML issues a warning. Since the default constructor does not appear in the text of the class, the warning message points to the class name.

B.11 `PossiblyNegativeIndex` warning

Array indices in array element access or assignment expressions must be non-negative values smaller than the size of the array. A `PossiblyNegativeIndex` verification failure is issued if OpenJML cannot prove that the index of an array access or assignment expression is non-negative.

Applying ESC to this example

```
public class Demo {

  public static int[] arr;

  //@ requires i < arr.length;
  static public int demo(int i) {
    return arr[i];
  }
}
```

results in this output:

```
Demo.java:7: warning: The prover cannot establish an assertion (
    PossiblyNegativeIndex) in method demo
    return arr[i];
               ^
1 warning
```

B.12 `PossiblyNegativeSize` warning

Java allows constructing new arrays with a run-time determined size, as in

```
int[] array = new int[x];
```

However, trying to create an array with a negative size will result in a run-time error (a `NegativeArraySizeException` exception). OpenJML issues a `PossiblyNegativeSize` verification failure if it cannot prove that the argument of an array allocation expression is non-negative.

Applying ESC to this example

```
1  public class Demo {
2
3    static public int[] demo(int i) {
4      return new int[i];
5    }
6
7    //@ requires i >= 0;
8    static public int[] demo2(int i) {
9      return new int[i];
10   }
11
12
13 }
```

results in this output:

```
1 Demo.java:4: warning: The prover cannot establish an assertion (
    PossiblyNegativeSize) in method demo
2     return new int[i];
3            ^
4 1 warning
```

B.13 `PossiblyTooLargeIndex` warning

Array indices in array element access or assignment expressions must be non-negative values smaller than the size of the array. OpenJML issues a `PossiblyTooLargeIndex` verification failure if it cannot prove that the index of an array access or assignment expression is less than the size of the array.

Applying ESC to this example

```
1  public class Demo {
2
3    public static int[] arr;
4
5    //@ requires 0 <= i;
6    static public int demo(int i) {
7      return arr[i];
8    }
9
10   //@ requires 0 <= i && i < arr.length ;
11   static public int demo2(int i) {
12     return arr[i];
13   }
14
15 }
```

results in this output:

```
1 Demo.java:7: verify: The prover cannot establish an assertion (
    PossiblyTooLargeIndex) in method demo
2     return arr[i];
3                ^
4 1 verification failure
```

In `demo2`, the range of the index is appropriately restricted so no verification failure is issued.

B.14 `Postcondition` warning

The `Postcondition` verification failure is issued when the postcondition of some behavior of the method is false. The postcondition is the conjunction, in order, of the *ensures* clauses of the behavior. There is a possible additional implicit postcondition that the result of of the method is non-null, if it is so declared (perhaps by default). If the precondition is not true for a behavior, then the postcondition need not be true. Postconditions apply only if the method terminates normally; they do not apply if the method ends with an exception, end with exiting the program (abruptly), or does not terminate at all.

This example shows a situation in which the implicit non-null-ness of the return

value is not established.

```
1  public class Demo {
2
3    static public void demo(int i) {
4      mm(i);
5    }
6
7    //@ requires i > 0;
8    //@ ensures \result == 1;
9    //@ also
10   //@ requires i == 0;
11   //@ ensures \result == 0;
12   //@ also
13   //@ requires i < 0;
14   //@ ensures \result == -1;
15   static Integer mm(int i) { // NonNull by default
16     if (i > 0) return 1;
17     if (i < 0) return -1;
18     return null;
19   }
20 }
```

The result of ESC is

```
1 Demo.java:18: warning: The prover cannot establish an assertion (
    Postcondition: Demo.java:15: ) in method mm
2     return null;
3     ^
4 Demo.java:15: warning: Associated declaration: Demo.java:18:
5   static Integer mm(int i) { // NonNull by default
6                  ^
7 2 warnings
```

B.15 `Precondition` warning

The `Precondition` verification failure is issued when the precondition of a method call is false. Note that the precondition being checked is the disjunction of the preconditions of all of the behaviors of the called method, including any inherited behaviors. That is, at least one of the behaviors must have a true precondition. The precondition of a behavior is the *conjunction* of the *requires* clauses, in order, of the behavior. There are also implicit requirements: any formal argument of a method that is a non-null reference type implicitly adds the

clause `requires` *arg* `!= null;` to each behavior.

```
1  public class Demo {
2
3    static public void demo(int i) {
4      mm(i);
5    }
6
7    //@ requires i > 0;
8    //@ ensures \result == 1;
9    //@ also
10   //@ requires i < 0;
11   //@ ensures \result == -1;
12   static int mm(int i) {
13     if (i > 0) return 1;
14     if (i < 0) return -1;
15     return i;
16   }
17 }
```

The result of ESC is

```
1 Demo.java:4: warning: The prover cannot establish an assertion (
    Precondition: Demo.java:10: ) in method demo
2     mm(i);
3       ^
4 Demo.java:10: warning: Associated declaration: Demo.java:4:
5   //@ requires i < 0;
6       ^
7 2 warnings
```

Note that when the precondition is the disjunction of multiple lines, the line reference in the message points to just one of them. It is important to not forget the other, especially inherited preconditions.

Check that all warnings are included; finish them all

Bibliography

[1] ANSI-C Specification Language. https://github.com/acsl-language/acsl/. 1, 3, 6

[2] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, New York, NY, 2003. 1, 3

[3] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. 6

[4] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, New York, NY, 2005. Springer-Verlag. 1, 3, 6

[5] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. Association for Computing Machinery. 1

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

[7] David R. Cok. The jSMTLIB User Guide, 2013. https://smtlib.github.io/jSMTLIB/. 51

[8] David R. Cok and Joseph R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2, including a case study involving the use of the tool to verify portions of an Internet voting tally system. In Gilles Barthe, Lilian Burdy, Marieke Huisman, Jean-Louis Lanet, and Traian Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS 2004)*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005. iii, 6

[9] David R. Cok, Gary T. Leavens, and Mattias Ulbrich. Java Modeling Language (JML) Reference Manual, 2nd edition, 2022. In progress. https://www.openjml.org/documentation/JML_Reference_Manual.pdf. 1, 3

[10] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, December 1998. 6

[11] Michael Ernst and students. The Checker Framework. https://checkerframework.org/manual/. 6

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

[13] Frama-C. https://frama-c.com. 6

[14] Patrice Godefroid. The soundness of bugs is what matters (position statement), 2005. https://alastairreid.github.io/RelatedWork/papers/godefroid:bugs:2005/. 90

[15] Michael Hicks. What is soundness (in static analysis)? http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/. 90

[16] Documentation for javac. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html#options. 29

[17] The KeY project. https://www.key-project.org. 3, 6, 21

[18] Gary T. Leavens. http://www.jmlspecs.org. 1, 3

[19] Gary T. Leavens, David R. Cok, and Amirfarhad Nilizadeh. Further Lessons from the JML Project, 2022. Accepted for publication. ii

[20] 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. 1, 5

[21] K. Rustan M. Leino et al. Dafny github site. https://github.com/dafny-lang/dafny. Accessed September 2021. 1, 3, 6

[22] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44?46, January 2015. 90

[23] Stainless verification framework. https://epfl-lara.github.io/stainless/intro.html. 1, 3

[24] Yichen Xie, Mayur Naik, Brian Hackett, and Alex Aiken. Soundness and its role in bug detection systems (position paper), 2005. https://alastairreid.github.io/RelatedWork/papers/xie:bugs:2005/. 90

Index