Does your software do what it should?

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

Updated to Java 21

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

DRAFT February 26, 2026

This document is being actively expanded, edited and reviewed.
Comments are welcome.

Foreword

OpenJML is the principal set of tools that work with the Java Modeling Language (JML). OpenJML combines an extended static checking tool (ESC), a runtime assertion checking compiler, and a few other tools for JML. OpenJML's ESC tool is the most powerful and useful automated verifier for current JML and current Java JML that is maintained as of this publication (in 2026). As such it is indispensable in working with JML.

I have worked with David Cok for over twenty years, since the time when we began work on the Java Modeling Language at Iowa State University, where I was a professor of Computer Science. He first started contributing to the Iowa State tools for JML that were developed by several of my PhD students at Iowa State University (notably Yoonsik Cheon, Clyde Ruby, and Curtis Clifton). At the time the Iowa State tools for JML were developed based on the Kopi Java compiler, which was an open source Java compiler.

He wrote the first version of the documentation tool, `jmldoc`, which displayed as a web page all the JML specifications for Java classes and interfaces and their methods.

After his work on the Iowa State tools, David worked with Joseph Kiniry to extend the ESC/Java tool from Digital Equipment Corporation to be fully integrated with JML and Java 4, producing ESC/Java2 [12].

However, as Java was rapidly evolving, it soon became clear to David and others in the JML community that it was difficult to keep tools for JML up to date with new Java features, particularly generics (introduced in Java 5). At the same time, support for the Kopi Java compiler was discontinued, leaving the Iowa State tools without a clear way to incorporate new Java features. Discussions among JML tool developers, including David, hit on the idea of basing the JML tools on the OpenJDK compiler, which was an open source version of Java that Sun (subsequently Oracle) guaranteed to maintain as the basis for new releases of Java, now produced by Oracle. This has indeed been the case, now at Java 21 in 2026. After some prototyping David decided that this was indeed a good idea, and he personally started the work to base tools for JML on the OpenJDK compiler, producing the OpenJML [8, 10, 11, 14] tool. Since then he has continued work on OpenJML as its principal developer.

David has also made many contributions to the design of JML itself. His insights are particularly appreciated, as they are based on his experience in consulting on specification and verification projects and his work on OpenJML. His contributions to Dafny and ACSL++ and his experience with Ada and Spec# have also influenced OpenJML and our joint work on JML. The JML community and myself have been fortunate to have David as a close colleague and continuing contributor over the past two and a half decades.

Dr. Gary T. Leavens
Professor of Computer Science
University of Central Florida
Orlando, Florida, USA May, 2022

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 [24] 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. OpenJML is conceptually built on ESC/Java, written at DEC SRC in the late 1990's [17], which Cok extended to ESC/Java2 [12] making the tool current with then-current Java and JML. Though OpenJML builds on the same concepts and technical decisions as ESC/-Java and ESC/Java2, it is a clean rewrite of the software. The incarnation of OpenJML described in this document is based on OpenJDK and is compatible with Java 21; previous versions have 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 document itself is 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 a 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/ [22] — including a book about KeY: https://www.key-project.org/thebook2/ [2]

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).
  - Spec# for the C# language,
  - and Dafny (https://github.com/dafny-lang/dafny).
- Previous work on JML tools preceding OpenJML, such as ESC/Java [17], ESC/-Java2 [12], and the ISU suite of tools [7].
- Occasional individual contributors to OpenJML itself.
- The OpenJDK compiler framework on which OpenJML is built: https://www.openjdk.org.
- Cross-fertilization with colleagues at the KeY project: https://www.key-project.org/.
- Contributed questions, bug reports and ideas, through industrial application, through undergraduate, master's and Ph.D. projects built on or using OpenJML, and through using OpenJML for classroom teaching.

Contents

Chapter 1

Introduction to JML and OpenJML

The Java Modeling Language [23] 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# [5] for C#, SPARK [3] for Ada, Stainless/Leon [6,29] for Scala, Dafny [26,27], and the KeY tool [22] for Java. JML has evolved considerably over the years, as Java has evolved. The JML Reference Manual (2nd edition) [13] is a substantial rewrite of the original Draft Reference Manual [25] 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 work, begun in 2023, supports Java 21. In addition there have been improvements in the ease of installing and running the tool. The source code, releases and development materials of OpenJML are hosted on GitHub,[1] 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.

[1] The previous project site, on Sourceforge, is obsolete and unmaintained.

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. The details of the JML definition are not repeated in this OpenJML user guide. 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://www.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, where correctness is important and worth extra effort. 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 software implementations against machine-readable specifications, stating what the software is expected to do. It requires the work of writing specifications in logical form along with the actual software implementation. However, 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 that are shown to agree must also represent what the software writer actually intended, but perhaps did not express completely or correctly, and that requires careful manual review of requirements that accompanies any 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) [13, 23]. As mentioned in the opening paragraphs of this chapter, there are other tools that perform the same task for other programming languages. OpenJML can also compile (like the Java compiler `javac`) Java source code, but including runtime-assertion checks (called RAC), which can add an additional level of verification to traditional dynamic testing.

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 OpenJDK 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 requirements 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 21, 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: unimplemented features of Java or JML are described in this document
- runtime checking of JML specifications: unimplemented features of JML are described in this document
- interacting with OpenJML programmatically from a host program is anticipated
- a language server for Java + JML is planned; such a language server would enable development of JML within a range of IDEs (a previous implementation of JML within Eclipse is currently obsolete and no longer supported)
- JML specifications included in javadoc documentation is 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 (tested on Ubuntu), 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: ESC/Java2 on a custom-built Java compiler; ISU tools were built 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, and 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 [25], which is being superseded by the document mentioned in the previous bullet

There are also other tools that use 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 at http://sourceforge.net/projects/jmlspecs 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 [15] and ESC/Java2 [12], and from discussions with builders of subsequent or concurrent tools such as Spec# [5], Boogie [4], Dafny [26], Frama-C [18], KeY [22], ACSL [1], and the Checker framework [16]. It was inspired by earlier work producing the Larch tools and, separately, the Eiffel language and tools. It also benefits from many advances in specification and reasoning 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 User Guide for OpenJML provides information on how to use OpenJML to do static and runtime verification for Java programs using JML specifications.

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 using WSL are supported.

- Find the latest release of the highest number series, currently 21, 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

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 (cf. §8).
- The executable `openjml-compile`, which is used to compile programs that programmatically call OpenJML (cf. §10).
- The executable `openjml-run`, which is used to run programs that programmatically call OpenJML (cf. §10).
- A `Solvers-...` folder containing executables for various SMT solvers used by OpenJML.
- The library `jmlruntime.jar`, which must be included with a RAC-compiled program when run with a conventional `java`. This library is included automatically when running a compiled program using `openjml-java`.
- The `specs` folder, which contains specifications for Java library (JDK) classes.
- 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.
- The `openjml.properties-template` file. (cf. §2.3)
- The `version-info.txt` file, which contains the version number and the github commit hashes of the version of the OpenJML sources that make up this release.
- The `jdk` folder. which contains the actual build.

What about the examples from this document

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. `javac` options (nearly) all lead with a single hyphen; OpenJML options typically begin with 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 (with a few exception)
- options and file paths may be freely intermixed on the command-line

3.2 Files and Folders

Besides options, the Java compiler allows only `.java` 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 `.java` 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. `.jml` files may not be listed on the command-line.

3.3 Output

OpenJML sends (nearly) 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. (The core `javac` emits some error messages to `System.err` .)

In addition, when operating as a compiler (e.g., for runtime assertion checking), OpenJML produces class files in the same manner as `javac` would, placing them either in the same folders as their corresponding `.java` files or in folders underneath the location indicated by the **-d** option.

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 symbolic names listed above are defined in `org.openjml.IAPI` and used when executing OpenJML programmatically (cf. §10).

The `EXIT_SYSERR`, `EXIT_ABNORMAL` and `EXIT_CANCELLED` values all indicate abrupt termination of OpenJML: there is a bug in the program or system resources have been exhausted or the program has been externally aborted. Any results from OpenJML are likely to be incomplete and maybe incorrect.

In normal operation, OpenJML may emit warning, error, and verification failure diagnostics.

- errors because of a faulty command-line will result in prompt termination of the program (without even any parsing of files), and a `EXIT_CMDERR` exit code.

An example is a misspelled option. Incorrect option values may not be detected until the option is actually used in OpenJML.

- other errors, such as parsing and type-checking errors, may allow processing of all the files listed on the command-line, but will stop before any compilation or deductive reasoning is attempted. These errors result in a `EXIT_ERROR` exit code.
- there may be warnings given, even if there are no errors. Warnings are typically lint-like advice, or statements that the program or command-line may be interpreted slightly differently than was intended. An example is the ignoring of some Java or JML feature because it is not executable or not yet implemented. The exit code is still `EXIT_OK` if there are warnings but no other errors or verification failures. As in the `javac` compiler, the -**Werror** option makes the presence of warnings an error itself.
- If OpenJML encounters no errors, it may proceed to deductive verification of the program and its specifications. If OpenJML fails to be able to prove any assertions, it will emit a verification failure. These are counted and recorded separately from warnings and errors. If there are verification failures, OpenJML will emit a `EXIT_VERIFY` exit code. That exit code can be changed by using the --**verify-exit** option; this option takes a integer value from 0–6 and it is that value that will be emitted as the exit code if there are any verification failures.

An OS shell typically considers any nonzero exit code as a command failure; that would include both errors and verification failures. If --**verify-exit** is used to set the exit code for veroification failures to 0, then verification failures would not be counted as command failures. In any case, the various exit codes provide the user the flexibility to write shell scripts to interpret the result is a variety of user-specific ways.

One other Java option, -**nowarn**, suppresses warnings (both OpenJDK and OpenJML warnings). -**nowarn** suppresses warnings before -**Werror** checks for them, so -**Werror** has no effect when -**nowarn** is used.

3.5 Other aspects of the environment

§5.1 describes using command-line options and Java properties to control the OpenJML tool. There are a few other aspects of the environment that do not typically affect users but are listed here for completeness.

Environment variables recognized by OpenJML Besides the environment variables that are interpreted as option settings, OpenJML recognizes the following environment variables. Note that they can be enabled just by setting them to an empty string, for example by `ERROR= openjml ....` None of these are ordinarily useful to users. The items labeled ‘Debugging’ may well change in the future.

DOuble check the following env vars

- **NOJML** – When set to a non-null value, the openjml executable is run in `-java` mode, that is, as just the native OpenJDK compiler. (This is needed as part of the OpenJDK compiler-build bootstrap process.)
- **OPENJML_ROOT** – Used to tell OpenJML where to find the solver executables and specification files. This is set automatically for standard installations and development environments.
- **ERROR** – Debugging: when set to a non-null value, a stack trace is printed along with each error message
- **VERBOSE** – equivalent to turning on `--jmlverbose` and `-verbose`
- **OJ** – Debugging: used to selectively enable various debugging output
- **STACK** – Debugging: enables printing stack traces for selected caught exceptions, particularly unexpected exceptions indicating internal bugs
- Environment variables whose names begin with **OPENJML_** are reserved to indicate values of command-line options

Names hard-coded in JML

- `.jml` – suffix for JML files
- `org.jmlspecs.annotation` – Java package name for JML annotations
- `org.jmlspecs.lang` – package name for various important JML entities (analogous to `java.lang`)
- `org.jmlspecs...` – general package prefix reserved for JML
- `org.openjml...)  - package prefix for the OpenJML programmatic API`

Names hard-coded in OpenJML

- `openjml`, `openjml-java`, `openjml-compile`, `openjml-run` – OpenJML executables in the installation folder
- `jmlruntime.jar` – The runtime-library needed when RAC-compiled programs are run using `java` (rather than `openjml-java`).
- `openjml.properties` – name of properties files for OpenJML
- `org.openjml...` – reserved prefix of property names used by OpenJML
- `OPENJML`, `ESC`, `RAC`, `DEBUG`, `KEY`, `KeY` – prefined conditional compilation keys (§4.5)

Furthermore, the OpenJML tool expects auxiliary material that it needs for proper operation to be in locations unchanged from those in the installation .zip file.

Chapter 4

OpenJML **Concepts**

4.1 Specifications in .java and .jml files

JML allows specifications for a Java class and its methods 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 compiled 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 that are not explicitly listed on the command-line; 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 classes that are defined in files not listed on the command-line. 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 a Java tool is 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 a source version available, the -**Xprefer** option (cf. 5.8) 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. 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.

The method of finding a specification file is best described for the following cases:

- for a `.java` file listed on the command-line, or not listed but referenced, perhaps recursively by classes on the command-line, the `.jml` file is sought in the following locations in turn
  - on the specspath
  - in the same folder as the `.java` file
  - otherwise use the `.java` file itself
- for a `.class` file with no corresponding `.java` file, the `.jml` file is sought
  - on the specspath
- for a `.class file` that is being used, but has a corresponding `.java` file on the sourcepath, the `.jml` file is sought
  - on the specspath
  - in the same folder as the corresponding `.java` file
  - otherwise use the `.java` file itself

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:
  - the argument to the command-line option --**class-path** (or an equivalent)
  - otherwise the value of the property `java.class.path`
  - otherwise the value of the system environment variable `CLASSPATH`
  - otherwise the default, which is the current working directory

  The system libraries are always appended to the classpath as set above.
  At run time only the classpath needs to be set; it is set to the package root of the compiled files using either the environment variable `CLASSPATH`, the command-line option `-Djava.class.path=...` or the command-line option `-cp ...`.
- `sourcepath`: The OpenJML sourcepath is set using one of these alternatives, in priority order:

  - the argument of the command-line option --**source-path** (or an equivalent)
  - 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 built-in location of the system library specifications always appended:
  - the argument of the OpenJML command-line option --**specs-path**
  - the value of the Java property `org.openjml.option.specs_path` (cf. the discussion concerning properties in §4.3)
  - 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 shadows 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-warn=missing-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 .jml file found on the specspath: if there is a .jml file in the same folder as the .java file, it is used; otherwise the .java file itself is used as the specification

- 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.

- 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: Are these two still correct

- 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
- a class file is used as the match for a given class, and there are two instances of a source file for the same class are on the sourcepath, with the one later in the sourcepath belonging to the class file

In these two scenarios, a `.java` file is used as the specification, but it is not the .java file that was used either as source or to compile the class file. OpenJML will likely issue error messages because of mismatched declarations 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

4.3.1 Properties and options

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 those options do not need to be repeatedly set on the command-line.

4.3.2 Finding properties files

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.

- A `openjml.properties` file in the OpenJML installation directory, 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
- Properties defined by environment variables, as described below, if any.
- Then the value of any property whose name has the form `org.openjml.option.`*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.

How does this work for the –warn/–no-warn and –infer/–no-infer options

4.3.3 --properties command-line option

One command-line option is the --**properties** option, which takes a path to a properties file as argument. This file is read and any options it defines are recorded *at its location in the command-line*. That is, any option in the file overrides previous command-line arguments and is overridden by any following command-line arguments. Also, then, the command-line may have multiple instances of the --**properties** option; each one is read and applied as it is encountered in the left-to-right parsing of the command line.

4.3.4 Interpreting properties files

These means of using properties apply only to OpenJML *options, not to OpenJDK options.* Note also that some command-line options are aliases for others. For example, --**esc** is an alias for --**command=esc**. In this case the alias (e.g., --**esc**) may not be set by a property; only the true option, in this case --**command**, may be set in a properties file.

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. Such whitespace is ignored.
- 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:

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

- `org.openjml.defaultProver` - the value is the name of the prover (cf. §4.4) to use by default
- `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.`*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 the names of 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 or unless the property definition is superseded by another value for --**command**.

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.)

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. That is why OpenJML needs to know the name of the solver and not just its executable, so that it can adjust its interaction with the solver accordingly.

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 `STRICT` is defined when the language choice (--**lang**) is `jml` and is not defined for other choices of language (the default language choice is `openjml`)
- the key `DEBUG` is reserved but is disabled by default
- the keys `KEY` and `KeY` are reserved for use by the KeY ([22]) tool and are 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 distinct from `KEY` .

Two simple uses are these:

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

The first line, with the + sign, is ignored in all situations except when `OPENJML` is defined as a key; this form might be used if the content of the comment is an OpenJML-specific extension. `KEY` might be used in a similar way.

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.

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

If there is no source file and no specification file for a class, then JML-defined default specifications are used. These are discussed in the JML Reference Manual (https://www.openjml.org/documentation/JML_Re

In addition, in some situations Java automatically generates methods and classes, such as default constructors and methods for user-defined `enum` and `record` classes. JML defines default specifications for these cases also.

[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.

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, allowing some checking for misuse of null references to be a type check rather than a proof obligation. Java itself has no provision for non-null types, but various tools (e.g., Checker framework [16]) 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 that 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 a 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 are (or were) in `javax.annotation`. 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. To remain sound, the return values of unspecified methods are still possibly-null; but the JML default for user-written code and specifications is that references are non-null. This policy helps identify null reference problems early and requires explicit specification that a reference may be null.

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 and uses 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 (or property definition), 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 §4.9.5.

In the future, nullness inference may be implemented, reducing the need for annotations on local variables and type parameters.

4.9.3 Nullness annotations for array declarations

Java type annotation syntax is 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 possibly-null reference to an array of non-null Strings.

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

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

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

`x` is a possibly-null reference to an array of non-null references to arrays of possibly-null Strings. In

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

`x` is a possibly-null reference to an array of possibly-null references to 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` .

For uninitialized declarations of array type, the elements of the initial value are all null, so the default type annotation for the elements is that they are nullable.

The implementation of nullness as type annotations is quite new, so it is still under experimentation, and some aspects may change in the future.

4.9.4 Nullness annotations for qualified class names

The syntax `@Nullable Object x` means that x may hold either `null` or an `Object`. But if one wants to use a fully-qualified class name, one must write `java.lang.@Nullable Object x` because it is the type name `Object` that is annotated with `@Nullable`, not the package name `java`. Similarly the annotation on a nested class name is written `MyClass.@Nullable Inner`.

4.9.5 Nullness for binary classes

For binary (.class) files with no source code and no explicit specifications, a default specifications are assumed. 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 visible class 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 Purity

JMLv2 introduced four different kinds of method purity (`pure`, `spec_pure`, `strictly_pure`, `no_state`) and changed the proof requirements and restrictions needed to use methods with these various purity markings in specifications. See the discussion in the language manual for more detail.

4.11 Arithmetic modes

4.11.1 Integer arithmetic

JML defines three arithmetic modes for integer arithmetic: bigint, safe, and java. The rationale and semantics for these are presented in the JML Reference Manual.

- 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 a 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 unbounded mathematical integers. This is the default for arithmetic in specifications.

The arithmetic 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 (and 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 (and in syntactically nested classes or interfaces).

The checks performed for arithmetic overflow are (by default) *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 §9.3.1). 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 in Java code, and line 4, because no constraints have been put on `i` and `j` by the previous failure on line 4; there is no overflow in the assertion because bigint mode is used in the specification, but the comparison in the assertion might fail.

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

One can also specify --**arithmetic-failure=quiet**. With this option, in safe mode, any arithmetic operations are assumed to be in range and there are no verification warnings issued.

4.11.2 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.

4.12 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. Also, bit-twiddling may be a more efficient way to implement some numeric operations. 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.

An individual method or class can be translated in a given mode by adding this annotation on the method or class (using `true` or `false` or `auto` as the argument): `@org.jmlspecs.annotation.Options("--esc-bv=true")`.

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.

4.13 Specification inference

Precise specifications can be verbose and writing them can be time-consuming. It 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 may 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 §12.1.

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. OpenJML has now added and prefers the two-hyphen, dash-case spelling of its options, with some 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 [21].

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>*	[§5.8] read options from a file. *This is implemented only for Java options, not* OpenJML *options*
-Akey	[§5.8] options to pass to annotation processors
--add-modules *<modulelist>*	[§5.11] see Java documentation re modules
-bootclasspath *<path>* --boot-class-path *<path>*	[§5.8] see Java documentation
-cp *<path>* -classpath *<path>* --classpath *<path>*	[§4.2, §5.8] location of input class files
-d *<directory>*	[§5.8] location of output class files
-deprecation	[§5.8] warn about use of deprecated features
--enable-preview	enables preview language features
-encoding *<encoding>*	[§5.8] character encoding used by source files
-endorseddirs *<dirs>*	[§5.8] see Java documentation
-extdirs *<dirs>*	[§5.8] see Java documentation
-g	[§5.8] generate debugging information
-h *<directory>*	location of generated header files
-? -help --help	[§5.6] output (Java and JML) help information
--help-extra	[§5.6] help about extra Java options
-implicit	[§5.8] whether or not to generate class files for implicitly referenced classes
-J*<flag>*	[§5.8] flags for the runtime system
--limit-modules *<modulelist>*	[§5.11] see Java documentation re modules
-m *<modulelist>* --module *<modulelist>*	[§5.11] see Java documentation re modules
--module-path *<path>*	[§5.11] see Java documentation re modules
--module-source-path *<path>*	[§5.11] see Java documentation re modules
--module-version *<version>*	[§5.11] see Java documentation re modules
-nowarn	[§5.9] show only errors, suppressing warning messages
-p *<path>*	[§5.11] like --module-path see Java documentation re modules
-parameters	see Java documentation
-proc	[§5.10] see Java documentation re annotation processing
-processor *<classes>*	[§5.10] see Java documentation re annotation processing
--processor-module-path *<path>*	[§5.10] see Java documentation re annotation processing
-processorpath *<path>* --processor-path *<path>*	[§5.10] where to find annotation processors see Java documentation re annotation processing
-profile	[§5.8] see Java documentation
--release *<release>*	[§5.7] target release for compilation
-s *<directory>*	[§5.8] location of output source files
-source *<release>* --source *<release>*	[§5.7] the Java version of source files
-sourcepath *<path>* --source-path *<path>*	[§4.2] location of source files
--system *<jdk>*	[§5.8] see Java documentation
-target *<release>* --target *<release>*	[§5.7] the Java version of the output class files
--upgrade-module-path *<path>*	[§5.11] see Java documentation re modules
-verbose	[§5.6] verbose output for Java compiler only, not OpenJML
-version --version	[§5.6] output (OpenJML) version
-Werror	[§3.4, §5.9] treat warnings as errors
-X	[§5.6] Java non-standard extensions

Options inherited from OpenJDK (cont.) See the Java documentation for more detail	
-Xlint	[§5.9] enable OpenJDK lint warnings

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

Options specific to JML Options listed as spelled with two initial hyphens may also be spelled with just one hyphen, with two preferred	
--arithmetic-failure *<mode>*	[§4.11.1] sets the mode for arithmetic checks: hard, soft (the default) or quiet
--check	[§5.3] typecheck only (**--command=check**)
--check-accessible -checkAccessible	[§7.3.5.1] whether to check accessible clauses (default: true)
--check-feasibility *<list>* -checkFeasibility *<list>*	[§7.2] kinds of feasibility to check (default: none)
--code-math *<mode>*	[§4.11] arithmetic mode for Java code (default: safe)
--command *<action>*	[§5.3] which action to do: `check` `esc` `rac` `compile` `parse` `doc`, default is `check`
--compile	[§5.3] typecheck JML but compile just the Java code (**--command=compile**)
--counterexample -ce	[§7.3.7] show a counterexample for failed static checks (default: off)
--defaults *<list>*	enables various default behaviors - *in development*
--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	[§4.12] whether to use bit-vector arithmetic (default: auto)
--esc-max-warnings *<n>* -escMaxWarnings *<n>*	[§7.1.1§7.3.6] max number of verification errors to report (default: unlimited)
--esc-warnings-path	[§7.1.1 §7.3.6] whether to find all failure paths to a failing assertion (default: off)
--exec *<file>*	[§7.3.3] file path to prover executable (default: installed executable)
--exclude *<patterns>*	[§7.3.4] paths to exclude from verification (default: no exclusions)
--extensions *<classes>*	[§11] comma-separated list of extensions classes and packages (default: no additional extensions)
--infer *<list>*	[§12.1] comma-separated list of keys designating specifications to infer if possible
--inline-function-literal	*Experimental*
-java	[§5.2] use the native OpenJDK tool, with no JML features (default: off)
-jml	[§5.2] process JML constructs (default: on)
--jmldebug	[§5.6] very verbose output (includes --progress) (**--verboseness=4**)
-jmltesting	[§5.6] changes some behavior for testing (default: false)
--jmlverbose	[§5.6] JML-specific verbose output (**--verboseness=3**)
--keys *<list>*	[§4.5] define keys for optional annotations (default: no keys)
--lang *<language>*	[§9] the JML variant to use (openjml (the default), jml)
--method *<patterns>*	[§7.3.4] methods to include in verification (default: all methods)

Options specific to JML (cont.) Options listed as spelled with two initial hyphens may also be spelled with just one hyphen, with two preferred	
--nonnull-by-default -nonnullByDefault	[§5.4] type uses are not null by default (default: on)
--normal	[§5.6] only outputs diagnostics; no other progress information (**--verboseness=1**)
--nullable-by-default -nullableByDefault	[§5.4] values may be null by default (default: off)
--os-name *<name>*	[§7.3.3] Operating System name to use in selecting prover (default: (auto), or one of `macos`, `linux`, `windows`)
--parse	[§5.3] parsing only (**--command=parse**)
--progress	[§5.6] outputs errors, warnings, progress and summary information (**--verboseness=2**)
--properties *<file>*	[§4.3.3] property file to read (no default)
--prover *<name>*	[§7.3.3] prover to use (default: z3-4.3)
--purity-check -purityCheck	[§6.3] check for purity in libraries (default: on)
--quiet	[§5.6] no output at all, just exit code (**--verboseness=0**)
--rac	[§5.3] compile runtime assertion checks (**--command=rac**)
--rac-check-assumptions -racCheckAssumptions	[§8.3.1] enables (default on) checking assume statements as if they were asserts
--rac-compile-to-java-assert -racCompileToJavaAssert	[§8.3.7] compile RAC checks using Java asserts (default: off)
--rac-java-checks -racJavaChecks	[§8.3.2] enables (default off) performing JML checking of violated Java features
--rac-missing-model-field-rep	[§8.3.9] controls behavior when model fields have no representation
--rac-precondition-entry -racPreconditionEntry	[§8.3.8] distinguishes precondition failures from external vs internal calls
--rac-show-source -racShowSource	[§8.3.6] includes source location in RAC assertion failure messages (values: `none`, `line`, `source`); default is `line`
--require-white-space	[§6.1] whether white space is required after an @ (default: false)
--show *<list>*	[§5.6] prints the details of source transformation (default: false)
--show-not-executable -showNotExecutable	[§8.3.4] warn about features not executable, in --rac operations (default: off)
--show-not-implemented -showNotImplemented	[§8.3.5] warn about features not implemented (default: off)
--show-skipped -skipped	[§7.3.4] show methods whose proofs are skipped (default: true)
--show-summary	[§7.3.9] shows a summary of numbers of methods and classes proved in one run of OpenJML (requires --progress)
--smt *filename*	[§7.3.11] where to write generated SMT files (for off-line use or inspection): default is no output
--solver-seed *<n>*	[§7.3.11] seed to pass on to the SMT solver (default: 0 - no seed)
--spec-math *<mode>*	[§4.11] arithmetic mode for specifications (default: bigint)
--specs-path *<paths>* -specspath	[§4.2] location of specs files
--split *<split-keys>*	[§9.3.6.6] splits proof of method into sections (default: no split)
-staticInitWarning	*Experimental*
--subexpressions	[§7.3.7] show subexpression detail for failed static checks (default: false)
--timeout *<seconds>*	[§7.3.11] timeout for individual prover attempts (default: no limit)
--trace	[§7.3.7] show a trace for failed static checks (default: false)
--triggers	enable SMT triggers (default: true)
-typeQuants	*Experimental*

Options specific to JML (cont.) Options listed as spelled with two initial hyphens may also be spelled with just one hyphen, with two preferred	
--verboseness <*n*>	[§5.6] level of verboseness (-1=silent, 0=quiet .. 4=jmldebug) (default: 1, **--normal**)
--verify-exit <*n*>	[§7.3.11] exit code for verification failures (default: 6)
--warn <*list*>	[§5.9] 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 white space must be enclosed in 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 are options and their values. 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 an 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.

- 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. If providing a value for the option is optional, then the value must be provided using the = syntax, not as an option and value separated by whitespace.

- 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. Most options do not accumulate values. These options may reasonably appear more than once in a command-line: --**dir**, --**dirs**, --**infer**, --**warn**, --**properties**.

- 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. Some option values are a string that is a comma-separated list of keywords. No white-space is permitted in such a list.

- 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. (This is the built-in default, before any settings from properties files, environment variables or the command-line.)

- 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 (but all the OpenJML options) may use the = form. The form --**xyz=** resets the option to its built-in default. (This is the built-in default, before any settings from properties files, environment variables or the command-line.)

5.2 Options: Operational modes

These operational modes are mutually exclusive.

- -**jml** (default, and by far the most common use) : 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; typically used just for testing that the OpenJML implementation has not changed OpenJDK's behavior on Java code
- -**java**: processes the command-line options and files using only OpenJDK functionality. No OpenJML functionality is invoked and no other OpenJML options are allowed. Any difference between this mode and the behavior of an unmodified OpenJDK implementation of `javac` is cause for investigation. If used, this option must be the very first option.

5.3 Options: JML tools

The following 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 **parse**, **check**, **esc**, **rac**, **compile**, **doc**. The default is to use the tool to do only typechecking of Java and JML in the source files (**check**).
- --**parse** : causes OpenJML to do only parsing of the Java input files and corresponding JML files (alias for --**command=parse**)
- --**check** : causes OpenJML to do only parsing and type-checking of the Java and JML in the input files and any referenced 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 (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**) *--doc is not yet implemented.*

5.4 Options: OpenJML options applicable to all OpenJML operational modes

- --**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*` would expand to all the java files in folders within 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*` there were no folders 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)
- --**lang** *language* : determines whether OpenJML extensions are allowed (the default)
- --**nullable-by-default** : sets the global default to be that all declarations are implicitly `@Nullable`, if they are not otherwise declared `@NonNull` (cf. §4.9)
- --**nonnull-by-default** : sets the global default to be that all declarations are implicitly `@NonNull` (the default), if not otherwise declared `@Nullable` (cf. §4.9)
- --**properties** *property-file* : provides a file from which to read properties as options
- --**show-not-implemented** : emits warnings about JML features that are ignored because they are not implemented; the default is enabled.(cf. §8.3.5)

5.5 Options: JSON and textual ASTs

OpenJML can emit a JSON representation of a compilation unit's (file's) parsed abstract syntax tree. By default, a `.java.json` file is produced for each `.java` file listed on the command-line. No references to other files are resolved, because the json files are emitted before any type or name resolution is performed.

Here is an example command-line:

```
openjml --parse --show=json A.java
```

JSON output with type information is produced with the option –**show=typed-json**.

The json output is still an experimental, work-in-progress, capability; feedback on its value and format is welcome. Eventually also it will be possible to deserialize a `.json` file back into a Java AST.

Reading json will be implemented in the future.

OpenJML can emit a simple tree-formatted textual representation of the parsed ASTs. To produce this output, use the options –**show=ast** or –**show=typed-ast**. The former produces output right after parsing; the latter produces output after type-checking and includes type information. This output is sent to `System.out`, and is primarily suitable for visual inspection for debugging.

5.6 Options: JML Information and debugging

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

- -**X**, --**help-extra** : Java option to print out help about advanced or experimental options
- -? , -**help**, --**help** : prints out help information about the command-line options
- --**help=warn**, --**help=infer** : print out help about warning and inference control (= form required)
- --**version** : prints out the version string of this build of the OpenJML tool

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

- --**quiet** : no output except the exit code
- --**normal** : (default) errors and warnings and verification failures and a summary of such problems; warnings can be omitted using -**nowarn** along with --**normal**.
- --**progress** : prints out progress 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). This is a single-hyphen Java option.
- --**jmlverbose** : prints out verbose information about the JML processing (includes --**verbose** and --**progress**)
- --**jmldebug** : prints out (voluminous) debugging information (includes --**jmlverbose**). The output is not at all guaranteed to be stable from version to version of OpenJML.

- --**verboseness** *<int>* : sets the verboseness level to a value from 0 .. 4, corresponding to --**quiet**, --**normal**, --**progress**, --**jmlverbose**, --**jmldebug**

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** or --**show=all** prints all four parts
  - --**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. is also used for this output, as discussed in §5.5:

  - --**show=json** – emits ASTs in JSON format for the parsed files, into files named `*.json` (cf. §5.5)
  - --**show=typed-json** – prints out ASTs with type information (after type-checking) for the parsed files in JSON format, into file (cf. §5.5)
  - --**show=ast** – prints out ASTs for the parsed files, to a text file with suffix `.ast`.
  - --**show=typed-ast** – prints out ASTs with type information (after type-checking) for the parsed files, , to a text file with suffix `.ast`.
- --**smt** *file-path* : emits the SMT input sent to the SMT solver into a file at the given location, as described in §7.3.11.

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**)

  The option also changes the output in the following ways, but only until the expected test outputs can be updated:

  - does not show location back-pointer information in 'Associated declaration' messages
  - does not verify class members in sorted order
  - does not emit information about bit-vector mode

  - in RAC, some location information is suppressed

5.7 Java Options: Version of Java language or class files

- --**source** *<level>* : this Java option specifies the Java version of the source files, with values of 4, ..., 21, ... This controls whether some syntax features are permitted. The default is the most recent version of Java (currently 21).
- --**target** *<level>* : this Java option specifies the Java version of the output class files (for compilation or RAC). The default is currently 21.

5.8 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 option value 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 option value 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)
- -**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, then: (a) if the option is 'source', always use the source file (recompiling the `.classs` file); (b) if the option is 'newer' then use whichever of the `.java` or `.class` file has a newer modification time. The default is 'newer'.

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**
- -**system**
- -**profile**
- -**J**
- -**X**... : Java's extended options

5.9 Control of lint-like warnings

OpenJDK allows enabling lint-like warnings, which come in a variety of categories. These are warnings about style or code constructions that, while not illegal, are questionable. They are mostly off by default and can be enabled using -**Xlint**, either by category or all of them with -**Xlint:all**. -**help-lint** gives a list of the warning categories.

OpenJML also has warnings. These are mostly on by default. Some warning categories are defined (and the rest are in process). The --**warn** option enables control of OpenJML warnings, by category.

All OpenJDK and OpenJML warnings can be disabled using the OpenJDK option -**nowarn**. Any OpenJDK and OpenJML warnings not disabled with -**nowarn** can be turned into errors using -**Werror** (cf. §??).

OpenJML warnings are grouped into categories; 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=none** — disable all warning categories
- --**no-warn=all** — disable all warning categories
- --**no-warn=none** — enable all warning categories
- --**warn=** — reset all warning categories to their defaults
- --**warn=reset** — reset all warning categories to their defaults
- --**warn=list** — emit help information listing each warning category, its current setting, and its default setting
- --**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

The command-line may contain multiple uses of the --**warn** and -**no**–**warn** options; each instance is applied as it is encountered, setting, unsetting, or listing the warning categories as appropriate.

*The set of implemented warning categories is under development. It may change and it may be unified with -**Xlint**. See --**warn-list** for a current list or --**help=warn**.*

5.10 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**

5.11 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**

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.

A set of Java files with JML annotations is parsed (only) with the command

```
openjml --parse options files
```

Performing only parsing is a rare occurrence, but may be useful to do initial syntax checking of a file before other related files that are needed for typechecking are available. The --**parse** command is also useful in conjunction with --**show=json** to produce JSON representations of a file's AST (cf. §5.5).

Parsing is affected by these options:

- -**classpath**, -**sourcepath** and --**specs-path** (§4.2)
- 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. Note that this option can only be applied to all the files being parsed, or to none of them; it cannot be applied on a file by file basis.

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

```
stlisting
or\\
\begin{lstlisting}[numbers=none,belowskip=-6pt, aboveskip=-6pt, emph={options,
    files}, emphstyle=\itshape, language=none, keywords={}]
openjml options 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** option can be used temporarily to 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 parsing and 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. §7.3.4). The ESC tool will result in one of five outcomes for each method:

- It finds no verification failures, because the specifications and program are consistent.
- It finds no verification failures because the program is infeasible.
- It issues one or more verification failure messages.
- 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 --**esc-max-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 is too complex for the tool to prove it.

For example, the program

```
public class MaxEscWarnings {

  public static void test(int i, int j) {
    if (i > j) {
      //@ assert j == 0;;
    } else {
      //@ assert i == 0;
    }
  }
}
```

and the command `openjml --esc MaxEscWarnings.java` produce the output

```
MaxEscWarnings.java:5: verify: The prover cannot establish an assertion (Assert) in method
     test
      //@ assert j == 0;;
          ^
MaxEscWarnings.java:7: verify: The prover cannot establish an assertion (Assert) in method
     test
      //@ assert i == 0;
          ^
2 verification failures
```

perhaps with the two errors in reverse order.

But with the command

```
openjml --esc --esc-max-warnings=1 MaxEscWarnings.java
```

only one error message is given (it could be either one).

By default, OpenJML produces a verification error message just once for a given assertion, even if it is reachable by multiple paths. In particular, the postconditions are single assertions reachable from each return statement. So with the command `openjml --esc MaxEscWarnings.java` the program

```
public class MaxEscWarnings {

  //@ ensures \result == 100;
  public static int test(int i, int j) {
    if (i > j) {
      return j;
    } else if (i < j) {
      return 100;
    }
    return i;
  }
}
```

produces the output

```
MaxEscWarnings.java:6: verify: The prover cannot establish an assertion (Postcondition:
     MaxEscWarnings.java:3:) in method test
      return j;
      ^
MaxEscWarnings.java:3: verify: Associated declaration: MaxEscWarnings.java:6:
  //@ ensures \result == 100;
      ^
2 verification failures
```

with just one verification error, even though the postcondition is violated by two return statements.

The option --**esc-warnings-path** (which is disabled by default) asks OpenJML to test each path to a given assertion, including each path to the postcondition clause. So the same program with the command

```
openjml --esc --esc-warnings-path MaxEscWarnings.java
```

gives the output

```
MaxEscWarnings.java:6: verify: The prover cannot establish an assertion (Postcondition:
     MaxEscWarnings.java:3:) in method test
      return j;
      ^
MaxEscWarnings.java:3: verify: Associated declaration: MaxEscWarnings.java:6:
  //@ ensures \result == 100;
      ^
MaxEscWarnings.java:10: verify: The prover cannot establish an assertion (Postcondition:
     MaxEscWarnings.java:3:) in method test
    return i;
    ^
MaxEscWarnings.java:3: verify: Associated declaration: MaxEscWarnings.java:10:
  //@ ensures \result == 100;
      ^
4 verification failures
```

(or some permutation of the given errors). The error report tells that two of the three return statements can violate the postcondition. To obtain more information about the input values that cause an assertion violation, use the --**trace** or --**counterexample** options (§7.3.7).

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, such as by breaking up larger methods into smaller pieces. 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`. *where* may be a comma-separated list of various location designators.

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

The question of feasibility is 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 successful (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 is 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

```
//CMD: 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:9: verify: There is no feasible path to program point at program
      exit in method T_Feasibility1.m(int)
  }
  ^
1 verification failure
```

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

```
//CMD: 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 `kk` 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 under inputs satisfying the preconditions). 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. However, sometimes dead code is purely innocuous. For example, if a precondition of a method contains a conditional statement and that method is called in a context where the condition always evaluates to true, say, then the false branch is always dead in that location.

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 `i == 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:

```
//CMD: 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
    }
  }

  //@ requires i != Integer.MIN_VALUE;
  //@ ensures \result >= 0;
  /*@ pure */ public static int abs(int i) {
    return i < 0 ? -i : i;
  }
}
```

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 thought to be 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:

```
//CMD: openjml --esc T_Feasibility3.java
```

```
public class T_Feasibility3 {

  //@ requires i >= 0;
  public void m(int i) {
    int j = abs(i);
    //@ show i, j;
    if (i != j) {
      // Should never get here!
      //@ unreachable
    }
  }

  //@ requires i != Integer.MIN_VALUE;
  //@ ensures \result >= 0;
  //@ pure
  public static int abs(int i) {
    return i < 0 ? -i : i;
  }
}
```

produces

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

which shows that that the verifier thinks that `i` and `j` can be different and thus that the `reachable` statement is reachable (the specific values of `i` and `j` may be different from run to run). Tightening up the specifications of `abs` will result in the expected behavior, as shown next.

```
//CMD: openjml --esc T_Feasibility3a.java
public class T_Feasibility3a {

  //@ requires i >= 0;
  public void m(int i) {
    int j = abs(i);
    //@ show i, j;
    if (i != j) {
      // Should never get here!
      //@ unreachable
    }
  }

  //@ requires i != Integer.MIN_VALUE;
  //@ ensures \result >= 0 && (\result == i || \result == -i);
  //@ pure
  public static int abs(int i) {
    return i < 0 ? -i : i;
  }
}
```

- Feasibility checking is not useful for runtime-checking. At runtime a program can only know that it has reached a particular program point; it cannot know whether other program points are reachable for other executions of a program.[1]

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

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 (the default)
- **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 JML `assert` statement; if the execution path to the assertion is not feasible, the assertion will never be checked
- **assume** – just after each explicit JML `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), including the implicit fall-through return
- **throw** – is every `throw` statement feasible (after computing the throw expression)
- **if** – are both branches of the `if` statement 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 and after inlining the specification summary
- **call** – before and after any call
- **halt** – at each JML `halt` statement
- **loopcondition** – at the beginning of the loop body
- **loopbody** – at the end of the loop body
- **loopexit** – on the exit branch after testing the loop condition and finding it false
- **loopcontinue** – at each `continue` statement in a loop body
- **break** – at a `break` statement in a block or a loop body
- **exit** - is it possible to exit the program (normally or with an exception) by any execution path (but before testing the postconditions)

There is one special case that must be used by itself: **preconditionOnly**. Checking feasibility with this alternative checks for feasibility of the combination of the invariants and the preconditions (like **precondition**), but then does not translate any of

use case is better served with a coverage checking tool.

the body of the method. This creates a much smaller verification problem.

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.

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 Inferring specifications

OpenJML includes some limited ability to infer specifications, controlled by the --**infer** option. The details are given in §12.1.

7.3.3 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 [9] 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 expressions, 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 [9], 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.

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

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. Though other provers have been evaluated, the only prover currently supported is `z3_4_3`, corresponding to Z3 version z3-4.3.1.
  expand the set of supported provers
- --**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, now obsolete, option related to solvers is --**logic**, whose value is the SMT logic to use or the word `ALL`. Current SMT solvers select a SMT logic automatically, typically all those logics needed to process the given SMT input. Thus this option is rarely needed and may well fail if actually tried. It is considered obsolete.

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

The default behavior is to check each method in each file and folder listed on the command-line. The set of methods checked can be constrained by the --**method** and --**exclude** 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. §9.4.1)

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 general rule for 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 (in quotes, to avoid the semicolon being interpreted by the shell).

- 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 entries 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. If none of the elements of the list contain parentheses, the list may be comma-separated.

The --**show-skipped** option controls output about which methods are being skipped for verification. Using this option (which is on by default, but is only applicable when the verbosity is at least --**progress**) 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 any one 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*`

7.3.5 Control over the checks performed

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.

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

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

However, specifying the reads footprint of a method is useful for pure methods that are used in specifications. With a small reads footprint, specified by an `accessible` (or, equivalently, `reads`) clause, OpenJML can know that the result of calling that method only depends on its arguments and the memory locations in the reads footprint. Particularly if the method does not depend on the heap at all, then reasoning about its behavior is much simplified.

Checking reads clauses consists of checking that any heap reference within a method, whether an explicit field or array element reference or implicitly by the reads clause of a called method, is a subset of the method's own reads clause. If this property cannot be established an `Accessible` verification failure will be reported. An example of such code is given in §B.3.

7.3.5.2 Checking purity: --check-purity

Ordinarily, OpenJML checks that only methods marked `pure` are used in specifications. However, many library methods are side-effect free and could be used in specifications, but do not have their own specifications written as yet. Using them in a specification would provoke an error. Turning off this option (`--no-check-purity`) disables this check for library methods (those in a package whose name starts with 'java').

Disabling this check (it is enabled by default) should be seen as a very temporary expedient. It is better to add the specifications for the needed methods. The behavior, and even existence, of this option may change in the future.

7.3.5.3 Checking static initialization: --static-init-warning

This option is under development. static-init-warning

7.3.6 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`)

- --**esc-warnings-path**: if enabled (disabled by default), then verification failures are reported for each execution path through a method to that assertion. If this option is disabled, only the first path found is reported. (Which path is found or the order in which multiple paths are found is non-deterministic.)

The above two options are described with examples in §7.1.1.

7.3.7 Counterexample information

When a prover is unable to prove an assertion, it often can display counterexample information giving specific values of source program variables that caused the assertion failure. The following options output some of that information.

- --**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

The output from these options is currently being reworked and improved to be more understandable. Currently, the best option to use is --**subexpressions**. but even that information is difficult to understand for anything but values of primitive types.

improve counterexample and tracing – Provide more information and examples

7.3.8 Dividing up the proof: --split

The `split` statement (§9.3.6) 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 §9.3.6.6.

7.3.9 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 subsection (§7.3.6).

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

- --**progress** : enables reporting the starting and completion of each method verification; these delineations also serve to associate warning and error reports with the method that produced them
- --**show-summary** : (default enabled) when enabled, and verboseness is at least `--progress`, then after all proofs have been attempted, a summary of proof successes, failures, timeouts, etc. is printed.

- **--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.

7.3.10 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.

- **--esc-bv** *mode* : this option controls whether fixed-bit-width integer operations are encoded as mathematical integer operations or as operations on bit-vectors. It is described in §4.12.

7.3.11 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 (so the solver picks some arbitrary, non-deterministic value).
- **--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`, which is simply a warning.

The value of -1 is also permitted as a legacy value for this option. With this value, a verification failure is reported exactly as a warning. That is, the diagnostic message will contain 'warning:' instead of 'verify:', there will be no separate summary of the number of verification failures (those will be included in the number of warnings), and the exit code will be 0 (if there are no errors).

- --**smt** *file-pattern* : This option gives the name 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. This use case is most appropriate when a single method is being verified. Alternatively, one 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**) means to not write any output file.

Chapter 8

OpenJML **tools — Runtime Assertion Checking**

In Runtime Assertion Checking (RAC), 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 classic dynamic testing.

8.1 Compiling classes with assertions

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 already compiled classes such as library 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 free of assertion violations.

Given a program with two files, `A.java` and `B.java`, with `A` referring to `B`, this compilation will cause both classes to be RAC-instrumented (presuming no `.class` files already exist): `openjml --rac A.java`

To compile `A` with RAC but not `B`, use these commands:

```
openjml --rac A.java
openjml --compile -Xprefer:source B.java
```

or, alternately,

```
openjml --compile B.java; openjml --rac A.java
```

It is helpful to understand what assertions are generated (and checked by RAC). 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, the compiler does not know whether both sides in the caller-callee relationship will be compiled with assertion checks; hence the precondition or postcondition is checked by both, to ensure it is at least checked once (cf. §8.3.1). The following list includes the most common checks; the full set is listed with more detail and examples in Appendix B.

- well-definedness checks of any JML expression, before the clause containing the expression is itself checked
- any explicit JML assert, reachable and unreachable statement
- any explicit JML assume statement (cf. §8.3.1)
- non-null checks when a object is deferenced (dot-operator or array-element operator)
- non-null checks when a reference variable or formal parameter or field declared NonNull is assigned or initialized
- array index is in range when an array is indexed
- preconditions and invariants of a callee, checked as assertions by the caller before calling a callee
- preconditions and invariants of a callee, checked as assumptions by a callee after being called but before executing the body of the callee (cf. §8.3.1)
- postconditions and invariants of a callee, checked as assertions by a callee after executing the body of the callee
- postconditions and invariants of a callee, checked as assumptions by a caller after returning from a callee (cf. §8.3.1)
- `assignable` and `accessible` clauses ane *not* checked by RAC, because that requires being able to test for freshness of allocations, which is not possible in general at runtime

Can we implement checking assignable and accessible and freshness?

8.2 Executing a 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 V21) but include the `jmlruntime.jar` library on the classpath.

8.3 Options specific to runtime checking

8.3.1 --rac-check-assumptions

When --**rac-check-assumptions** is enabled (the default), both assumptions and assertions are checked. One situation in which this is helpful is the checking of explicit `assume` statements. Such statements can be a guide to static deduction by stating conditions that might be difficult to prove. When checked using runtime checking, a test of the `assume` statement will show if that assumption is false for any test case in the runtime test suite.

Checking both assumptions and assertions 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

```
public class A {

  public static void main(String ... args) {
    m(args.length);
    mm(args.length);
  }

  //@ requires i == 1;
  //@ ensures \result == 20;
  public static int m(int i) {
    return 10;
  }

  //@ requires i == 0;
  //@ ensures \result == 20;
  public static int mm(int i) {
    return 10;
  }
}
```

when compiled with the command

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

and run with

```
openjml-java A
```

produces the output

```
A.java:4: verify: JML precondition is false
    m(args.length);
     ^
A.java:10: verify: Associated declaration: A.java:4:
  public static int m(int i) {
                    ^
A.java:8: verify: JML precondition is false
  //@ requires i == 1;
      ^
A.java:16: verify: JML postcondition is false
  public static int mm(int i) {
                    ^
A.java:15: verify: Associated declaration: A.java:16:
  //@ ensures \result == 20;
      ^
A.java:5: verify: JML postcondition is false
    mm(args.length);
      ^
A.java:15: verify: 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. The postcondition of `m` is not checked because its precondition is not true.

However, if the example is compiled with

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

the output is

```
A.java:4: verify: JML precondition is false
    m(args.length);
     ^
A.java:10: verify: Associated declaration: A.java:4:
  public static int m(int i) {
                    ^
A.java:16: verify: JML postcondition is false
  public static int mm(int i) {
                    ^
A.java:15: verify: 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, we would want both the postconditions and preconditions checked by the caller, and then we would want this option enabled.

8.3.2 --rac-java-checks

When the option --**rac-java-checks** is enabled (default is disabled), 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
`ArrayIndexOutOfBoundsException` 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: verify: 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. Java's report of an exception is sent to the standard error output (`System.err`), whereas JML's output goes to standard out (`System.out`).

This behavior also interacts with the `allow` and `forbid` features of JML (see the JML reference manual [13]). For example, placing an

```
//@ allow ArithmeticException;
```

annotation on the line containing the potential divide by zero tells OpenJML that the user expects to catch the `ArithmeticException` if one is thrown from this statement and so OpenJML should not issue a JML warning, despite any request to do so using `--rac-java-checks`. So this source text

```
public class A {

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

}
```

compiled with the command

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

and run with

```
openjml-java A
```

yields

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

8.3.3 --nonnull-by-default and --nullable-by-default

These options apply to runtime checking in the same way as they do to static checking. See the discussion in §7.3.1.

8.3.4 --show-not-executable

When enabled (default is disabled), **--show-not-executable** 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 RAC 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. It may also be helpful to comment out features that are not executable using the `//-RAC@` conditional JML comment (§4.5).

8.3.5 --show-not-implemented

When enabled (default is disabled), option--show-not-implemented warns about the use of features that are not yet implemented (and thus ignored). The user may wish to enable this option in order to be alerted to features that are being ignored because they are not yet implemented. Some features may be implemented for static checking but not for RAC (or vice versa). It may also be helpful to comment out features that are not implemented for RAC using the `//-RAC@` conditional JML comment (§4.5).

8.3.6 --rac-show-source

--rac-show-source *choice* (default: line; choices: none, line, source) tells OpenJML to include source location information 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 including the 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’ or even to ‘none’.

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=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
```

8.3.7 --rac-compile-to-java-assert

When enabled (default is disabled), **--rac-compile-to-java-assert** compiles RAC checks into Java `assert` statements (which throw `java.lang.AssertionError` when enabled using **-ea** during execution, instead of using `org.jmlspecs.runtime.JmlAssertionError`. When this option is enabled, all assertion violation reporting is through Java `assert` statements and all the alternatives described in §8.4 are ignored. Furthermore, no reports will be generated at all at runtime unless the Java option **-ea** is enabled. 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.)

So this input file

```
public class A {

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

when compiled with the command

```
openjml --rac --rac-compile-to-java-assert A.java
```

and run with

```
openjml-java -ea A
```

produces the output

```
Exception in thread "main" java.lang.AssertionError: A.java:4: verify: JML assertion is false
    //@ assert args.length == 1;
        ^
        at A.main(A.java:4)
```

Whereas when compiled with the command

```
openjml --rac --no-rac-compile-to-java-assert A.java
```

and run with

```
openjml-java A
```

it produces the output

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

8.3.8 --rac-precondition-entry

In automated testing using RAC, it is useful to distinguish precondition errors at the top-level, which indicate invalid input data, from internal precondition errors, which indicate some bug in using a method internally. The **--rac-precondition-entry** option (off by default) enables such behavior.

Here is an example:

```
import org.jmlspecs.runtime.*;
public class Demo {

  //@ requires i >= 0;
  public static void outer(int i) {
    inner(i);
  }

  //@ requires i > 0;
  public static void inner(int i) {
  }

  public static void main(String ... args) {
    outer(1); // OK
    try {
      outer(-1); // Outer precondition fails
    } catch (JmlAssertionError.PreconditionEntry e) {
      System.out.println("OK");
    }
    try {
      outer(0); // Inner precondition fails
    } catch (JmlAssertionError.PreconditionEntry e) {
      throw e; // FAILURE;
    } catch (JmlAssertionError.Precondition e) {
      System.out.println("OK");
    }
    try {
      inner(0); // Precondition fails
    } catch (JmlAssertionError.PreconditionEntry e) {
      System.out.println("OK");
    }
  }
}
```

If the above program is compiled with

```
openjml -rac -rac-precondition-entry PreconditionEntry.java
```

and run with `openjml-java PreconditionEntry`, then the output is just a series of “OK" text. Inspecting the program shows that

- the `main` method acts as test harness and calls `outer`, which calls `inner`, with various arguments.
- when `outer` is called with an argument that violates its own precondition, a `JmlAssertionError.PreconditionEntry` exception is thrown
- when `outer` is called with an argument that satisfies its own precondition but violates the precondition of the call to `inner`, a `JmlAssertionError.Precondition` exception is thrown
- however, if `inner` is called directly with an argument that violates its own precondition, a `JmlAssertionError.PreconditionEntry` exception is thrown

Using this option sets the `-Dorg.jmlspecs.openjml.rac=exception` alternative described in §8.4. It does not work if **--rac-compile-to-java-assert** is set. It works whether or not **--rac-check-assumptions** is set.

8.3.9 --rac-missing-model-field-rep

JML permits classes to declare *model fields* as a specification abstraction mechanism. Model fields may have no explicit representation, with their behavior specified axiomatically by invariants and other specifications. Alternately, a model field may be given an explicit representation using a `represents` clause, but that clause may be in a derived class and may be separately compiled.

In RAC, a model field is compiled as a generated method that can be overridden by whichever class actually supplies an implementation of the field. However, it might well be that no representation is ever given. This situation is not a problem for ESC.

For example, the following code

```
public class Demo {

  //@ model public int modelField;
  //@ represents modelField = 42;

  public static void main(String... args) {
     Demo d = new Demo();
     //@ show d.modelField;
  }

}
```

compiled and run with
`openjml-compile --rac Demo.java; openjml-run Demo`
gives this output:

```
LABEL JMLSHOW_1 = 42
```

But if the `represents` clause is missing, the output becomes

```
Demo.java:2: warning: JML model field does not have a representation: modelField
  //@ model public int modelField;
                       ^
Demo.java:6: warning: JML ignoring statement because model field does not have a
     representation: Demo.modelField
      //@ show d.modelField;
               ^
2 warnings
```

This behavior can be modified a bit. If compiled with
`openjml-compile --rac --rac-missing-model-field-rep=fail Demo.java`
the warning becomes an error:

```
Demo.java:2: error: JML model field does not have a representation: modelField
  //@ model public int modelField;
                       ^
1 error
```

And if compiled with
`openjml-compile --rac --rac-missing-model-field-rep=zero Demo.java`
a generated implementation is created that returns a zero-equivalent value:

```
Demo.java:2: warning: JML substituting zero-equivalent representation because model field
     does not have a representation: modelField
  //@ model public int modelField;
                       ^
1 warning
LABEL JMLSHOW_1 = 0
```

The value of the option **–rac-missing-model-field-rep** may also be

- `zero-quiet`, which is like `zero`, but gives no warning message
- `skip`, which is the same as with no option given (that is, `skip` is the default option value)
- `skip-quiet`, which just silently skips a statement containing an unimplemented model field

8.4 Controlling how runtime assertion violations are reported

There are several ways in which a RAC-compiled program can report assertion violations, in addition to using Java assert statements as described in §8.3.7. These can be controlled by a property set at the time the RAC-enabled program is *run* (not when it is *compiled*). Thus all of these options use the same compiled class files. Assertion violations can be reported ...

- A) as messages printed to `System.out`, if property `org.jmlspecs.openjml.rac` is not set or set to the value `stdout`. In this case the program will continue executing after printing the assertion violation and may possibly encounter and report additional violations or Java exceptions.
- B) as messages are issued as in (A) above, but a stack trace is printed along with the error message, if `org.jmlspecs.openjml.rac` is set to `showstack`. This makes the output more verbose, but may make it easier to debug why a particular violation is occurring.
- C) as a thrown exception of some subtype of `org.jmlspecs.utils.JmlAssertionError`, if the system property `org.jmlspecs.openjml.rac` is set to `exception`. The subtype is determined by the kind of violation, as described later in this section. Execution of the program stops with the first violation reported, unless the `JmlAssertionError` is caught and handled.
- D) as a thrown exception of the type `java.lang.AssertionError`, if `org.jmlspecs.openjml.rac` is set to the value `assertionerror`. Execution of the program stops with the first violation reported unless this `AssertionError` is caught and handled. This is the same kind of assertion that is thrown by a Java `assert` statement, but does not depend on using the **-ea** option.
- E) as a thrown exception of the type `java.lang.AssertionError` but generated

from a Java `assert` statement, if `org.jmlspecs.openjml.rac` is set to the value `javaassert`. It is required that the option **-esa** is used at runtime also. Note that the option **-esa** is required, rather than **-ea**.[1]

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

```
openjml-java -Dorg.jmlspecs.openjml.rac=stdout MyProgram
```

As an example, the input file

```
public class A {

  //@ requires i > 0;
  public static void m(int i) {
  }
  public static void main(String ... args) {
    m(0); // Precondition error
  }

}
```

when compiled with the command

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

and run with

```
openjml-java A
```

or

```
openjml-java -Dorg.jmlspecs.openjml.rac=stdout A
```

(option A) produces the output

```
A.java:7: verify: JML precondition is false
    m(0); // Precondition error
     ^
A.java:4: verify: Associated declaration: A.java:7:
  public static void m(int i) {
                     ^
A.java:3: verify: JML precondition is false
  //@ requires i > 0;
      ^
```

If compiled the same way but run with

```
openjml-java -Dorg.jmlspecs.openjml.rac=showstack A
```

(option B) the output is

```
org.jmlspecs.runtime.JmlAssertionError$Precondition: A.java:7: verify: JML precondition is false
    m(0); // Precondition error
     ^
A.java:4: verify: Associated declaration: A.java:7:
  public static void m(int i) {
                     ^
        at java.base/org.jmlspecs.runtime.Utils.createException(Utils.java:127)
        at java.base/org.jmlspecs.runtime.Utils.assertionFailureL(Utils.java:96)
        at A.main(A.java:7)
org.jmlspecs.runtime.JmlAssertionError$Precondition: A.java:3: verify: JML precondition is false
```

[1] I believe this difference is because when checks are generated as in §8.3.7, the Java assert statements are generated directly in user code, hence the option **-ea** is used. But in the use case described here, the Java assert statement resides in library code that is part of the `java.ase` module, and hence the **-esa** option is needed.

```
  //@ requires i > 0;
      ^
        at java.base/org.jmlspecs.runtime.Utils.createException(Utils.java:127)
        at java.base/org.jmlspecs.runtime.Utils.assertionFailureL(Utils.java:96)
        at A.m(A.java:3)
        at A.main(A.java:7)
```

If compiled the same way but run with

`openjml-java -Dorg.jmlspecs.openjml.rac=exception A`

(option C) the output is

```
Exception in thread "main" org.jmlspecs.runtime.JmlAssertionError$Precondition: A.java:7:
     verify: JML precondition is false
    m(0); // Precondition error
     ^
A.java:4: verify: Associated declaration: A.java:7:
  public static void m(int i) {
                     ^
        at java.base/org.jmlspecs.runtime.Utils.createException(Utils.java:127)
        at java.base/org.jmlspecs.runtime.Utils.assertionFailureL(Utils.java:94)
        at A.main(A.java:7)
```

If compiled the same way but run with

`openjml-java -Dorg.jmlspecs.openjml.rac=assertionerror A`

(option D) the output is

```
Exception in thread "main" java.lang.AssertionError: A.java:7: verify: JML precondition is
     false
    m(0); // Precondition error
     ^
A.java:4: verify: Associated declaration: A.java:7:
  public static void m(int i) {
                     ^
        at java.base/org.jmlspecs.runtime.Utils.assertionFailureL(Utils.java:99)
        at A.main(A.java:7)
```

Finally, if compiled the same way but run with

`openjml-java -esa -Dorg.jmlspecs.openjml.rac=javaassert A`

(option E) the output is

```
Exception in thread "main" java.lang.AssertionError: A.java:7: verify: JML precondition is
     false
    m(0); // Precondition error
     ^
A.java:4: verify: Associated declaration: A.java:7:
  public static void m(int i) {
                     ^
        at java.base/org.jmlspecs.runtime.Utils.assertionFailureL(Utils.java:101)
        at A.main(A.java:7)
```

Generally speaking, mechanism (A) or (B) is the easiest and most useful. However, mechanism (C) is useful for fine-grained control over which assertions are reported. Different types of violations have different *labels*, such as `Invariant` or `Precondition`. 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 a runtime violaton corresponding to the given label occurs. 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 (cf. §8.3.8).

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 may or may not be 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) or (B) in §8.4). 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

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

when compiled with the command

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

and run with

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

produces the output

```
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 text is output, and the exit code is 0.

8.6 RAC FAQs

This 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.

For example, with `P.java`

```
public class P {

//@ model public static int i;
//@ static represents i = 10;

}
```

and `T.java`

```
public class T extends P {

  public static void main(String... args) {
    //@ assert i == 11;
  }
}
```

then executing the commands

```
openjml --rac T.java
openjml -Xprefer:source --compile P.java
openjml-java T
```

produces the result

```
Exception in thread "main" java.lang.NoSuchMethodError: ’int P.‘model$i()’
        at T.main(T.java:4)
```

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: boolean 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.

These warnings are not issued everywhere they should be. More warnings

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 that 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** and --**no-java-checks** options.
- Use --**rac-show-source=line**.

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 from standard JML, 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 Syntax

9.1.1 Optional terminating semicolons

JML uses semicolons to terminate statements and clauses. When a JML clause is immediately followed by the end of Java comment, with no immediately consecutive JML comment, then it is clear that the JML clause is ended. In those situations, the terminating semicolon is optional.

For example, in the code snippet

```
j = 1;
//@ assert j == 1;
j = 2;
```

If the semicolon terminating the `assert` statement is omitted, it triggers only a warning, not an error.

However, if there is a following JML statement, as in

```
j = 1;
//@ assert j == 1;
//@ unreachable
j = 2;
```

then the `assert` statement requires a semicolon to separate it from the `unreachable` statement.

9.2 Completeness and disjointness clauses

Grammar:

```
<behaviors-clause> ::= behaviors
        ( complete | local_complete
        | disjoint | local_disjoint );
```

One or more `behaviors` clauses may be placed at the end of a method specification. They are not part of any specification case. Rather they assert relationships among the specification cases; these assertions are checked during both static checking and runtime checking. Each such clause consists just of the `behaviors` keyword, a second keyword, and a terminating semicolon. Those second keywords are one of the following:

- `complete` – asserts that the preconditions of all of specification cases, including those inherited from super classes and interfaces are together complete, that is, that their disjunction can be proved to be `true`.
- `local_complete` – asserts that the preconditions of all of specification cases of the method as declared within the method's containing class, *but not* those inherited from super classes and interfaces, are together complete, that is, that their disjunction can be proved to be `true`.
- `disjoint` – asserts that each pair of preconditions taken from all of specification cases, including those inherited from super classes and interfaces, are mutually disjoint, that is, that the conjunction of each pair can be proved to be `false`.
- `local_disjoint` – asserts that each pair of preconditions taken from the specification cases declared within the method's containing class, *but not* those inherited from super classes and interfaces, are mutually disjoint, that is, that the conjunction of each pair can be proved to be `false`.

These checks are not performed if there are no `behaviors` clauses.

In runtime-checking, the check is just for the specific set of method parameters. That is, the behaviors are considered `disjoint` if, for the given parameters, at most one behavior has a true precondition; the behaviors are considered `complete` if, for the given parameters, at least one behavior has a true precondition. Note that if no preconditions are true, that is, not complete, then the composite precondition is false also, causing a precondition verification failure.

The completeness checks can be a sanity check and a guard against maintenance errors. Disjointness checks guard against pairs of specification cases not being disjoint and then having conflicting postconditions.

This feature is inspired by the corresponding feature in ACSL [1].

If there is enough use and demand, this clause can be extended to take a list of identifiers of specification cases and then apply the completeness and disjointness tests to that subset of specification cases.

9.3 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 v2):

- assert statement
- assume statement
- unreachable statement
- local ghost variable and model class declarations
- set statement
- block specifications
- loop specifications

OpenJML adds these, described in succeeding subsections:

- check statement (§9.3.1)
- show statement (§9.3.2)
- havoc statement (§9.3.4)
- halt statement (§9.3.5)
- split statement (§9.3.6)
- reachable statement (§9.3.7)
- use statement (§9.3.8)
- inlined_loop statement (§9.3.9)
- comment statement (§9.3.10)

9.3.1 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.

9.3.2 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 and a counterexample is available, then OpenJML emits messages giving 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, other than on the `System.out` output stream..

9.3.3 print statement

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

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

The print statement is ignored by static checking; when compiled for runtime-assertion checking, it prints the values of the arguments, space-separated, to System.out. It enables output when running the program without altering the actual Java program and without placing a burden of verifying Java output calls.

9.3.4 havoc statement

Grammar:
<havoc-statement> ::=
havoc *<opt-name>* *<storeref-expression>* ... ;

The havoc statement includes a list of *<storeref-expression>*s, 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.

Here is a simple example. In this code, the assert statements all succeed, except the last one. Note that we cannot simply omit the initializers in the variable declarations because Java requires all variables to be initialized before they are used; that rule applies to uses in JML statements as well.

```
public class Demo {

    public static void main(String... args) {
        int i = 0;
        short s = 0;
        //@ assert i == 0;
        //@ havoc i, s;
        //@ assert Integer.MIN_VALUE <= i <= Integer.MAX_VALUE;
        //@ assert Short.MIN_VALUE <= s <= Short.MAX_VALUE;
        //@ assert i == 0; // FAILS
    }
}
```

In RAC, a havoc statement is ignored, though havoced memory locations may in the future be initialized with some arbitrary value.

9.3.5 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. If the `halt` statement is not in a branch, then the postconditions are elided also.

For example, running --**esc** on this code

```
public class Demo {

  //@ ensures false;
  public static void main(String ... args) {
    if (args.length == 1) {
        //@ assert false;
    }
    //@ halt;
    if (args.length == 2) {
      //@ assert false;
    }
  }
}
```

produces

```
Demo.java:6: verify: The prover cannot establish an assertion (Assert) in method main
        //@ assert false;
            ^
1 verification failure
```

Because processing stops with the `halt` statement, no verification errors are produced for the second `assert` statement or the postcondition.

`halt` statements are ignored when running RAC.

9.3.6 `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. 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 that abbreviates the block when the rest of the method body is verified. Statement specifications cause splits into subproofs without an explicit `split` statement.

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)
- 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 `if` statement is an if-elseif-else chain, with the statement comprising the else branch being an `if` statement itself (and not a block) then the `split` annotation on the outer `if` statement applies to the whole chain.
- 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. For and for-each and while and do-while loops can all be split.
- If the `split` is a standalone statement with a boolean expression, two subproofs are constructed, one in which the expression is true and one in which it is false.

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 three consecutive if-statements, each preceded by a `split` statement, eight different verification conditions will be created. On the other hand, if one 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.

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

Here are some short examples. Use the command

```
openjml --esc --progress Demo.java
```

to see the different proofs.

9.3.6.1 If split

Demo file:

```
public class Demo {

  public static void test(int i) {
    //@ split
    if (i > 0) {
      //@ assert i > 1; // ERROR
    } else if (i < 0) {
      //@ assert i < -1; // ERROR
    } else {
      //@ assert i == 0; // OK
    }
  }
}
```

The output is

```
Proving methods in Demo
Starting proof of Demo.Demo() with prover !!!!
Completed proof of Demo.Demo() with prover !!!! - no warnings
Starting proof of Demo.test(int) with prover !!!!
Proof attempt for split A
Demo.java:6: verify: The prover cannot establish an assertion (Assert) in method test
      //@ assert i > 1; // ERROR
          ^
Result of split A is Not verified
Proof attempt for split BA
Demo.java:8: verify: The prover cannot establish an assertion (Assert) in method test
      //@ assert i < -1; // ERROR
          ^
Result of split BA is Not verified
Proof attempt for split BB
Result of split BB is Verified
Composite result Not verified
Completed proof of Demo.test(int) with prover !!!! - with warnings
Completed proving methods in Demo
2 verification failures
```

There are three subproofs, having the identifiers **A**, **BA** and **BB**.

9.3.6.2 Switch split

Demo file:

```
public class Demo {

  public static void test(int i) {
    //@ split
    switch (i) {
      case 0:
        //@ assert i > 1;
        break;
      case 1:
        //@ assert i == 1;
        break;
      default:
        //@ assert i == 0;
    }
  }
}
```

The output is

```
Proving methods in Demo
Starting proof of Demo.Demo() with prover !!!!
Completed proof of Demo.Demo() with prover !!!! - no warnings
Starting proof of Demo.test(int) with prover !!!!
Proof attempt for split A
Demo.java:7: verify: The prover cannot establish an assertion (Assert) in method test
        //@ assert i > 1;
            ^
```

```
Result of split A is Not verified
Proof attempt for split B
Result of split B is Verified
Proof attempt for split C
Demo.java:13: verify: The prover cannot establish an assertion (Assert) in method test
        //@ assert i == 0;
            ^
Result of split C is Not verified
Composite result Not verified
Completed proof of Demo.test(int) with prover !!!! - with warnings
Completed proving methods in Demo
2 verification failures
```

A subproof is attempted for each of the three cases of the switch statement.

9.3.6.3 Loop split

Demo file:

```
public class Demo {

  //@ requires 0 <= i;
  public static void test(int i) {
    int k = 0;
    //@ maintaining 0 <= k <= i;
    //@ decreases i-k;
    //@ loop_assigns k;
    //@ split
    while (k < i) {
      k++;
      //@ assert 0 <= k <= i; // OK
      //@ assert k == 0; // ERROR
    }
    //@ assert k == i; // OK
    //@ assert k == 10; // ERROR
  }
}
```

The output is

```
Proving methods in Demo
Starting proof of Demo.Demo() with prover !!!!
Completed proof of Demo.Demo() with prover !!!! - no warnings
Starting proof of Demo.test(int) with prover !!!!
Proof attempt for split A
Demo.java:13: verify: The prover cannot establish an assertion (Assert) in method test
      //@ assert k == 0; // ERROR
          ^
Result of split A is Not verified
Proof attempt for split B
Demo.java:16: verify: The prover cannot establish an assertion (Assert) in method test
    //@ assert k == 10; // ERROR
        ^
Result of split B is Not verified
Composite result Not verified
Completed proof of Demo.test(int) with prover !!!! - with warnings
Completed proving methods in Demo
2 verification failures
```

9.3.6.4 Boolean split

Demo file:

```
public class Demo {

  public static void test(int i) {
    //@ split i == 1;
    //@ assert i > 0; // OK for one split, not for the other
  }
}
```

The output is

```
Proving methods in Demo
Starting proof of Demo.Demo() with prover !!!!
Completed proof of Demo.Demo() with prover !!!! - no warnings
Starting proof of Demo.test(int) with prover !!!!
Proof attempt for split A
Result of split A is Verified
Proof attempt for split B
Demo.java:5: verify: The prover cannot establish an assertion (Assert) in method test
    //@ assert i > 0; // OK for one split, not for the other
        ^
Result of split B is Not verified
Composite result Not verified
Completed proof of Demo.test(int) with prover !!!! - with warnings
Completed proving methods in Demo
1 verification failure
```

9.3.6.5 Statement spec split

Demo file:

```
public class Demo {

  //@ requires 0 <= n < 100;
  //@ ensures \result == n*(n-1)/2;
  public static int test(int n) {
    int sum = 0;
    //@ refining
    //@  assigns sum;
    //@  ensures sum + sum == n*(n-1);
    {
    //@ maintaining 0 <= i <= n;
    //@ maintaining sum + sum == i * (i-1);
    //@ loop_assigns sum;
    //@ decreases n-i;
    for (int i=0; i<n; i++) {
      sum += i;
    }
    }

    //@ assert sum == n*(n-1)/2;
    return sum;
  }
}
```

The output is

```
Proving methods in Demo
Starting proof of Demo.Demo() with prover !!!!
Completed proof of Demo.Demo() with prover !!!! - no warnings
Starting proof of Demo.test(int) with prover !!!!
Proof attempt for split A
Result of split A is Verified
Proof attempt for split B
Result of split B is Verified
Composite result Verified
Completed proof of Demo.test(int) with prover !!!! - no warnings
Completed proving methods in Demo
```

Here the ‘B’ proof is the proof of the body of the statement specification, that is the loop. The ‘A‘ proof summarizes the block containing the loop with just the statement specification, which states what changed in the block and what the result of the block’s computation is (i.e. the postcondition) and goes on to include (and verify) the rest of the method body.

9.3.6.6 The --split option

Each subproof is given a designator consisting of a sequence of uppercase letters. For example an if-else-statement each branch of which contained another if-else-statement would spawn four subproofs 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`. These designators can be used with the --**split** command-line option.

For example, the if-split example above (§9.3.6.1) shows three subproofs, labelled A, BA and BB. The command

```
openjml --esc --split=BA,BB Demo.java
```

will attempt just the second and third of these subproofs.

The argument of the --**split** option is a comma-separated list of such proof designators. If the argument is an empty string, then all subproofs are attempted.

The negated form, --**no-split**, takes no argument and disables all the split statements, so that the method proof is attempted with one large verification condition. The split statements are always disabled for RAC.

If there is no --**split** or --**no-split** command-line option, then all subproofs are attempted in turn if the method contains any `split` statements.

The split option can be applied method-by-method using the `@Option` modifier, described in §9.4.5.

9.3.7 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:

```
void m1(int i) {
   //@ assert i == 0; // ERROR: i can be any integer,
                      //        not just 0
}
void m2(int i) {
   if (i > 0) {
     //@ reachable // OK - reachable in some scenario
   }
}
//@ requires i > 0;
void m3(int i) {
   if (i < 0) {
     //@ reachable //ERROR: not reachable
                  // with precondition and if condition
  }
}
```

The reachable statement is only used when 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.*

```
//@ requires i > 0;
void m(int i) {
   if (i > 0) { ... }
   else {
     //@ reachable
     throw new RuntimeException("Argument not positive");
  }
}
```

Reachability checking is discussed in more detail in §7.2.

9.3.8 use statement

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

Type information: The expression is typically a method call with a void return.

The use statement is a proof aid. It states a lemma that the prover then can use to prove subsequent code.

Here is an example using a method call.

```
public class Demo {
  //@ public normal_behavior
  //@   requires p >= 0;
  //@   ensures (p&1) ==  p%2;
  //@ spec_pure
  public void mod2lemma(int p) {}

  //@ requires k >= 0;
  //@ requires k <= Integer.MAX_VALUE/2 && k >= Integer.MIN_VALUE/2;
  public void mm(int k) {
    //@ show k;
    k = 2*k;
    //@ use mod2lemma( (k+1) );
    boolean b = ((k+1)&1) == 1;
    //@ assert b;
  }
}
```

The code above contains two methods. One is named `mod2lemma`. It is a method that simply has a precondition and postcondition. The body is empty. Because there is a body, OpenJML (under `--esc`) will attempt to prove that the postcondition follows from the precondition. For complicated lemmas, the method body might contain assertions that can coax the prover to a proof. This lemma states an equivalence between an arithmetic operation (mod by 2) and a bit-vector operation (bit-and with 1). This lemma proves quickly enough but often equivalences between arithmetic and bit-vector operations can be very time-consuming to prove. And of course, the content of a lemma may be any useful intermediate proposition that is convenient to prove as a separate step.

The second method uses this lemma in its body to establish an assertion about a computation. The `use` statement simply invokes the lemma *with specific arguments*. This is equivalent to instantiating the lemma were it expressed as a universally-quantified logical formula.

The `use` statement is then equivalent to asserting that the lemma's precondition holds for the given arguments and then assuming that the postcondition holds for those arguments, neatly making use of what was separately proved.

In this example, both methods are automatically verified, using the command `openjml --esc --progress Demo.java`

Note that the precondition must be proved to hold for the given arguments. If that proof fails a `UndefinedLemmaPrecondition` warning is reported. An example of such a failure is given in §B.50.

TODO - other kinds of use statement, and examples; discuss substitution

9.3.9 `inlined_loop` statement

Grammar:

<jml-inlined-loop-statement> `::=`
    `inlined_loop ;`

The `inlined_loop` statement is an experimental statement used to assist in the proofs of situations in which a called routine has a specification containing a model program that contains a loop.

more about inlined_loop, including examples; needs grammar with loop specs

9.3.10 comment statement

Grammar:

```
<jml-comment-statement> ::=
        comment <opt-name> <jml-literal-expression> ;
```

The comment statement takes a single expression that must be a string literal. The effect of the statement is to insert in the translated code a (Java) comment containing the given string. This can be useful when inspecting the output of the --**show** option (§5.6). It is purely a debugging aid. The string literal may be a concatenation (with +) for compile-time constant strings performed by the compiler.

9.4 Modifiers

9.4.1 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.

9.4.2 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 (just for verification, not for compilation). 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.

- the `inline` modifier is only applicable in ESC; it is ignored for RAC and every other operational mode
- the callee method (the one marked `inline`) is still verified as usual, ignoring the `inline` modifier; if it has no explicit specifications, it is verified against the usual default specifications; a method marked `inline` typically, but not necessarily, has no explicit specification
- the caller checks any explicitly given specifications of the callee (such as explicit `requires` clauses) but also inlines the body of the callee; the effect is that the body serves as a model program (though it is interpreted as Java code, not JML statements)
- An `inline` method must be `final`; a `final` modifier may be added in JML.

For example, with the code

```
public class Demo {

  //@ inline final pure
  public void callee(int i) {
    //@ assert i > 0;
  }

  public void caller() {
    callee(10);
    callee(-10);
  }
}
```

OpenJML produces the output

```
Demo.java:5: verify: The prover cannot establish an assertion (Assert) in method callee
    //@ assert i > 0;
        ^
Demo.java:5: verify: The prover cannot establish an assertion (Assert: Demo.java:10:) in
     method caller
    //@ assert i > 0;
        ^
Demo.java:10: verify: Associated declaration: Demo.java:5:
    callee(-10);
          ^
3 verification failures
```

The first verification failure message is the failure to verify the assert when checking the `callee` method. The second failure indicates a failure of the assert statement when it is inlined in place of `callee(-10)`, as indicated by the 'Associated declaration' message. There is no failure of the assert statement when it is inlined in place of `callee(10)`. Without the `inline` modifier, `caller` would verify without error.

9.4.3 query and secret

These are experimental modifiers in OpenJML used to specify observational purity and hidden state.

Probably write a full concept section on observational purity

9.4.4 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`?
- How does immutability interact with observational purity?

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

9.4.5 `@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=basic")`.

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:

```
//@ @Options("--esc-max-warnings=1")
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, --**specs-path** is used during parsing and typechecking, and so would not be allowed to be applied to a method in this way, but --**esc-bv=false** can be.

9.4.6 Experimental modifiers

Some other modifiers are under discussion, but not yet implemented in OpenJML. These include

- two_state
- captured
- infer (or something like it)

9.5 Expressions

9.5.1 \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

9.5.2 Enhancements to conditional annotations: \key

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

- In expressions, the term \key(id), is either a true or false Boolean literal, depending on whether the given *id* is defined as a conditional annotation key or not. The *id* is a comma-separated list of simple identifiers and string literals, though almost always just one. If there is more than one, the expression is true iff all of the identifiers are defined. The keys here are the same keys as are used in §4.5, defined in the same way with the --**keys** option. A \key expression may be combined in larger expressions just like any other boolean subexpression. The substitution of a boolean literal for the \key term takes place during parsing.

Here is an example of the \key notation:

```
public class Demo {

  public static void test() {
    //@ assert \key("OPENJML",ESC); // OK when run in OpenJML and --esc
    //@ assert \key(A); // OK if A is defined as a key, ERROR otherwise
  }
}
```

The command openjml --esc Demo.java produces

```
Demo.java:5: verify: The prover cannot establish an assertion (Assert) in method test
    //@ assert \key(A); // OK if A is defined as a key, ERROR otherwise
        ^
1 verification failure
```

However, the command openjml --esc --keys=A Demo.java verifies both assert statements.

9.5.3 \choosex quantified expression

Grammar:

```
<jml-choosex-expression> ::=
       \choosex <type-name> <java-identifier> ;
       [ [ <jml-expression> ]; ] <jml-expression>
```

JML has the `\choose T x; R(X); V(x)` expression; it returns some value `x` of type `T` for which `R(x) && V(x)` is true.

OpenJML adds an extension, the `\choosex T x; R(X); V(x)` expression; it returns the value of `V(x)` for some `x` for which `R(x)` is true.

Like `\choose`, `\choosex` is deterministic, though the value chosen to be returned is arbitrary. For a `\choosex` expression to be well-defined, there must exist an `x` satisfying `R(x)`.

9.6 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> ... ;
```

Need work here

9.7 Other topics to include, possibly

This is a list of various topics to be discussed, eventually:

- reasoning about captured objects (including `capture` modifier)
- `non_null_elements`
- adding new specification types
- post for old/pre declarations in specifications
- \nonnullelements for collection classes
- specification of lambda functions
- begin end markers
- inlined_loop
- \values
- \reach
- multiple arguments for \invariant_for, \static_invariant_for

- invariants method spec clause
- use of for_example as feasibility; also feasibility_behavior
- recommends-else, requires-else, also-else
- expanded array-range syntax; store-refs that include expressions
- functional form of \lbl
- \same

Chapter 10

Using OpenJML and OpenJDK within user programs

10.1 Executing openjml

OpenJML can be executed programmatically. Here is a sample program:

```
import java.io.*;
import org.openjml.IAPI;
public class Run {
  public static void main(String... args) {
    PrintWriter pw = new PrintWriter(System.out);
    int ex = -1;
    try {
      ex = IAPI.openjml("--esc", "A.java");
      System.out.println("EXIT: " + ex);
      ex = IAPI.openjml("--rac", "A.java");
      System.out.println("EXIT: " + ex);
      IAPI api = IAPI.make();
      ex = api.execute("--check", "B.java");
      System.out.println("EXIT: " + ex);
      if (ex != IAPI.OK) System.out.println("Exit not OK");
    } finally {
      pw.close();
    }
  }
}
```

The program can be compiled and run with these commands:

```
openjml-compile Run.java
openjml-run Run
openjml-java A
```

Here `A.java`, `B.java` and `Run.java` are expected to be in the current directory, and `openjml-compile` and `openjml-run` must be on your executable `PATH` or replaced by filepaths to the locations of those scripts in the installation directory.

The effect of the three commands above is as follows:

- line 1 simply compiles the `Run.java` program; no OpenJML functionality is executed.
- line 2 runs the compiled program. This causes three things to happen, as seen in the body of `Run.java`:
  - runs `--esc` on `A.java`
  - runs `--rac` on `A.java`, which produces a compiled `A.class`
  - runs type-checking (`--check`) on `B.java`
- then line 3 executes the RAC-instrumented, compiled class `A`

The `openjml-compile` executable compiles a program, like `javac`, except that it includes the openjml-enhanced jdk code and relevant options about modules to make it convenient to compile programmatic calls to `openjml`. Similarly, `openjml-run` is like `java`, but adds in the relevant options for using `openjml` code programmatically. `openjml-java` is also like `java`, but only adds in the runtime-environment needed to execute the rac-compiled class (here `A`).

In the `Run.java` program, there are two calls that execute the same logic as the `openjml` command-line. The first, `IAPI.openjml` (a static method), runs to completion and retains no intermediate state. The second uses an instance of `IAPI`, created by `IAPI.make()`, and then calling the non-static method `execute`.

The `IAPI` instance will eventually allow inspection of intermediate results such as type-checked ASTs corresponding to the program. Different context objects can be used to execute separate compilations, even concurrently (though this has not been tested, nor has the OpenJDK code been reviewed for thread-safety.)

Note that `IAPI` declares some `int` constants that are the values of the exit code from running `IAPI.openjml` or api.execute. The symbolic names are `OK`, `ERROR`, etc. as documented in §3.4. An example use is shown in the code snippet above.

Java 9 introduced modules to the Java language. OpenJDK 21, on which OpenJML is built, heavily uses modules to constrain access to internal aspects of the OpenJDK compiler. Consequently, exposing internal data structures as one would like in a programmatic API is more challenging than with non-module versions of OpenJDK. So, the programmatic API is in progress.

10.2 Redirecting output

The default behavior is for error and warning messages to be printed to `System.out` . This behavior can be changed in two ways: (a) by designating alternate output `PrintWriters` and (b) by designating a `DiagnosticListener` (see the following section).

Alternate output destinations are designated when the API context is created with the `IAPI.make` call. The no-argument version of this method simply designates de-

fault behavior. Instead one can designate a `PrintWriter` to use for normal output and a separate one to use for error and warning messages. The full signature of `IAPI.make` is `IAPI.make(PrintWriter out, PrintWriter err, DiagnosticListener<? extends JavaFileObject> listener)`,
with an additional convenience signature: `IAPI.make(PrintWriter out, \DiagnosticListener<? extends JavaFileObject> listener)`, which
uses the same `PrintWriter` for both outputs. Any of the parameters may take a `null` value, which means to use the default behavior.

10.3 Collecting Diagnostics

One can also register a (just one) listener for diagnostics. Such a listener implements the interface `DiagnosticListener`, which has just one method: `report(Diagnostic diagnostic)`. That method is called whenever a diagnostic is issued by OpenJML.

An example listener is shown below:

```
import org.openjml.*;
import javax.tools.*;

class Listener implements DiagnosticListener<JavaFileObject> {
  @Override
  public void report(Diagnostic<? extends JavaFileObject> diagnostic) {
    System.out.println("DIAGNOSTIC REPORTED");
    System.out.println("    Kind:           "
                      + diagnostic.getKind());
    System.out.println("    Source:         "
                      + diagnostic.getSource());
    System.out.println("    Start position: "
                      + diagnostic.getStartPosition());
    System.out.println("    Position:       "
                      + diagnostic.getPosition());
    System.out.println("    End position:   "
                      + diagnostic.getEndPosition());
    System.out.println("    Line number:    "
                      + diagnostic.getLineNumber());
    System.out.println("    Column number:  "
                      + diagnostic.getColumnNumber());
    System.out.println("    Message:        "
         + diagnostic.getMessage(java.util.Locale.getDefault()));
  }
}
```

This listener can be combined with this class:

```
import org.openjml.*;
import javax.tools.*;

public class Run {

  public static void main(String... args) {
    try {
      IAPI api = IAPI.make(null, new Listener());
      int x = api.execute("--esc","--progress","-jmltesting","A.java");
      System.exit(x);
    } catch (Exception e) {
      System.out.println("Exception: " + e);
    }
  }
}
```

When compiled and run using
`openjml-compile Run.java`
`openjml-run Run`
the following output is produced:

```
Proving methods in A
Starting proof of A.A() with prover !!!!
Completed proof of A.A() with prover !!!! - no warnings
Starting proof of A.m() with prover !!!!
Completed proof of A.m() with prover !!!! - no warnings
Starting proof of A.n() with prover !!!!
DIAGNOSTIC REPORTED
    Kind:           MANDATORY_WARNING
    Source:         SimpleFileObject[.../examples/apilistener/A.java]
    Start position: 110
    Position:       110
    End position:   110
    Line number:    7
    Column number:  9
    Message:        The prover cannot establish an assertion (Postcondition) in method n
DIAGNOSTIC REPORTED
    Kind:           MANDATORY_WARNING
    Source:         SimpleFileObject[.../examples/apilistener/A.java]
    Start position: 58
    Position:       58
    End position:   58
    Line number:    5
    Column number:  9
    Message:        Associated declaration
Completed proof of A.n() with prover !!!! - with warnings
Completed proving methods in A
```

10.4 Tokenizing text

The programmatic API allows tokenizing text, producing a standalone sequence of tokens. Because OpenJDK's `Token` class is not readily accessible outside of OpenJDK, the API here returns `WrappedTokens`, from which information about the underlying token can be obtained.

Here is an example of using the iterator. This input text:

```
import org.openjml.*;

public class Run {

  public static void main(String... args) {
    try {
      IAPI api = IAPI.make();
      IAPI.ITokenIterator iter =
          api.makeTokenIterator(args[0]);
      while (iter.hasNext()) {
        var t = iter.next();
        System.out.println(t + " : " + t.pos() + " "
          + t.endPos() + " " + t.kind() + " "
          + t.jmlKind() + " " + t.getTokenClass());
        System.out.println("    " + t.toStringDetail());
      }
      System.out.println("DONE");
    } catch (Exception e) {
      System.out.println("EXCEPTION: " + e);
    }
  }
}
```

compiled with the commands

```
openjml-compile Run.java
openjml-run Run "public class A { }"
```

produces the output

```
public : 0 6 public null class com.sun.tools.javac.parser.Tokens$Token
    [public:0:6]
class : 7 12 class null class com.sun.tools.javac.parser.Tokens$Token
    [class:7:12]
A : 13 14 token.identifier null class com.sun.tools.javac.parser.Tokens$NamedToken
    [token.identifier:13:14:A]
’{’ : 15 16 ’{’ null class com.sun.tools.javac.parser.Tokens$Token
    [’{’:15:16]
’}’ : 17 18 ’}’ null class com.sun.tools.javac.parser.Tokens$Token
    [’}’:17:18]
token.end-of-input : 18 18 token.end-of-input null class com.sun.tools.javac.parser.
     Tokens$Token
    [token.end-of-input:18:18]
DONE
```

10.5 Access to parsed ASTs

This capability and other such functionality is in development.

Add more API stuff

Commented out material to remove?

Chapter 11

Extending OpenJML

This chapter is draft material under development.

11.1 Basic Concepts

11.2 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 the classpath. 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).

say more about the code organization

11.3 Adding command-line options

11.4 Adding modifiers

11.5 Adding statement specification clauses

11.6 Adding method specification clauses

11.7 Adding class specification clauses

11.8 Adding built-in types

11.9 Adding new AST nodes

11.10 Adding new compiler phases

Chapter 12

Other OpenJML tools

12.1 Inferring specifications

This section will expanded in the future.

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

12.1.1 The --infer option

Specification inference is controlled by the --**infer** command-line option. Each different inference capability is controlled by a *key*, which is a simple predefined text string. Each capability can be either on or off; some are by default on and others by default off. Instances of this option on the command-line all build on each other, in order, which is different behavior than most options.

- --**help=infer** – emits brief help information about the option
- --**infer=**<*comma-separated-list-of-keys*> – enables all the capabilities named in the comma-separated list
- --**no-infer=**<*comma-separated-list-of-keys*> – disables all the capabilities named in the comma-separated list
- --**infer=list** – emits text stating all the implemented options, whether each one is currently enabled, and whether it is enabled by default
- --**infer=all** – enables all implemented capabilities
- --**infer=none** – disables all implemented capabilities
- --**infer=reset** – sets all implemented capabilities to their default values

- **--infer=show**, **--no-infer=show** – turns on (or off, respectively) emitting informational messages that state the inferred specifications (this is on or off for all the inference capabilities as a group). The default is off.

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

12.1.2 `loop_assigns` clauses

Key: **loop-assigns**
Default: Enabled

The `loop_assigns` clause, if absent from a loop specification, is inferred by analyzing the pattern of assignments in the loop body. The JML identifier `\count` is always implicitly included. Every loop must have either an explicit or an inferred `loop_assigns` clause.

Provide examples?

12.1.3 invariants describing static final fields

This inference capability is not yet controlled by the infer option

Java can have fields that give symbolic names to compile-time constants. In fact, such declarations are encouraged, so as to avoid 'magic numbers' in the source code. These fields are typically `public`, `static` and `final`; they do not change after a class has completed its static initialization. There is thus an implicit invariant that each such field maintains its initial value.

It is a nuisance and verbose to have to state this fact as an invariant in each class. Thus for each class OpenJML infers a static invariant that states that each public, static, final field with a compile-time constant initializer has its initial value after static initialization is complete. Each class will also include in its local invariant the corresponding invariant from its superclass and super interfaces.

Note that the inferred invariant only includes fields that are `public`, `static`, `final` and are initialized with a compile-time constant expression.
(cf. https://docs.oracle.com/javase/specs/jls/se21/html/jls-15.html#jls-15.29 for details.)

12.2 Generating Documentation

This section will be added later.

12.3 Generating Specification File Skeletons

This section will be added later.

12.4 Generating Unit Test framework

This section will be added later.

12.5 Generating Test Cases

This section will be added later.

12.6 Symbolic Execution and Abstract Interpretation

This section will be added later.

12.7 On-line sandbox for JML and OpenJML

This section will be added later.

A website that enables experimenting with JML and OpenJML can be more convenient than needing to install and setup a whole Java/JML/OpenJML development environment. Such websites have been created in the past but are no longer maintained.

12.8 Language server

This section will be added later.

Language servers have become common for providing support for a given language (here Java+JML) and related tools (OpenJML) in the context of an IDE, such as Eclipse, emacs, VSCode or IntelliJ.

Chapter 13

Limitations of OpenJML's implementation of JML

13.1 Soundness and Completeness

Much is made of the soundness and completeness claims of program analysis tools. In fact program 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. 13.1 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. 13.1 to be empty, ideally. They are 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 [19, 20, 30]. 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).

[1] Gary leavens suggested this approach.

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

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 trade-off 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* [28]: 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 then not 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: the tool may contain bugs that lead to missing actual errors. And little of sophisticated program analysis tools are actually verified themselves.

13.2 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.

13.2.1 Non-conservative defaults

- Methods are assumed to be deterministic (the same result is produced for the same arguments in the same state); any non-determinism must be explicitly specified. The most conservative default would be non-determinism. However, deterministic behavior is by far more common and certainly generally expected by users.
- Constructors are assumed to set only their own fields.

13.2.2 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.

13.2.3 Numeric and bit-vector arithmetic

OpenJML does not handle arbitrary mixtures of integer and bit-vector operations. Such a situation does produce warnings.

13.2.4 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.

13.2.5 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`.

13.2.6 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`).

13.2.7 Reflection

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

13.2.8 Class loading

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

13.2.9 Hash codes

The `hashCode` of an `Object` is not necessarily deterministic; if an `Object` allocation follows some `ghost` code execution, the `hashCode` value may depend on whether that `ghost` code was executed or not. Consequently it is possible for RAC code to affect the values of `hashCodess`, and therefore to affect how objects are stored in hash-maps and hash-sets. None of this dependency should affect the external behavior of such collections.

Furthermore, two objects of the same type that are not `equal` may have equal or unequal hashCodes. Whether those hashCodes are equal or unequal may be affected by the execution history. No program should rely on consistency of such a relationship and no proof system should be able to prove anything about such relationships.

13.2.10 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.

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

13.3.1 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`
- `\not_assigned`
- `duration`
- `working_space`
- `\duration`
- `\working_space`
- `\space`

13.3.2 Termination

OpenJML does prove termination of loops, using the `loop_decreases` clause. It also checks termination of simple direct recursion using the `measured_by` clause. Some JML extensions to the well-founded measure used in a `measured_by` clause are needed to handle general recursion.

13.3.3 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.

13.3.4 Arithmetic mode

- OpenJML does not implement `code_bigint_math`
- OpenJML does not consistently implement floating point mathematics
- Versions of Java prior to v17 defined a `strictfp` keyword. JML did not define its behavior and OpenJML did not implement anything. As of v17, that keyword is obsolete; all floating point semantics must adhere to the IEEE 754 standard.

13.3.5 Quantifiers

OpenJML does not support the `\sum`, `\product`, and `\num_of` quantifiers in ESC; it does not support the set comprehension expression in either ESC or RAC.

RAC supports quantified expressions only in those cases having a range expression that can be transformed into an iteration; often this means that quantified expressions with multiple declared variables are better expressed as nested quantified expressions.

13.3.6 Static initialization

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

13.3.7 model import static statement

OpenJML does not properly implement `model import static` statements. It does properly implement non-static model import statements, although it did not do so until recently.

In these two examples,

```
import static a.X;
//@ model import static b.X;
```

```
import static a.*;
//@ model import static b.X;
```

two classes named `X` are imported by the combination of 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, `model import static` statements are ignored.

13.3.8 Model programs

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

- the `extract` modifier and clause
- the `choose` clause
- the `repeat` clause
- the `returns` clause
- the `breaks` clause
- the `continues` clause

13.3.9 Universe types

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

Other stuff not implemented yet: record type, pattern matching in switch statements and expressions

Chapter 14

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).

14.1 GitHub

The GitHub project named OpenJML (https://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
  - `OpenJML21`
  - `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.
- `OpenJMLDemo`: demo material for 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 or archived):

- `OpenJML-UpdateSite`: the update site for the Eclipse plug-in
- `OpenJMLFeature`: an Eclipse feature project implementing OpenJML as a plugin to Eclipse
- `SMTSolvers`: an Eclipse feature plug-in containing the Solvers project, so the solvers can be distributed through an update site
- `jdk8u-dev-langtools`, `jdk8u-dev-langtools-old-mirror`: obsolete snapshots of the OpenJDK8 sources
- try-openjml
- openjml-installer

14.2 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 related 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.

14.3 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 there, sufficient to allow new developers to create a correct development environment, run tests, and create releases on GitHub, etc.

14.4 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.

14.5 Creating and using a development environment

14.5.1 Setup

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

14.5.2 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 folder with the source code:

- `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

A build of openjml produces the OpenJML executable and the runtime library, `jmlruntime.jar`, which are referred to by a number of scripts in the release package.

The build procedure for OpenJDK is quite complex. It uses a slightly older version of OpenJDK to do a bootstrap build of OpenJDK; it also uses some C/C++ code (for which it needs the clang compiler).

14.6 Running tests

The tests are organized as unit and functional tests in the `OpenJMLTest` folder. The procedure for running tests is maintained here: https://github.com/OpenJML/OpenJML/wiki/OpenJmlTesting.

14.7 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.

In brief, the release procedure has these steps:

- Build current versions of the documentation pdfs or the JML Reference manual and this Users' Guide. Also assemble the tutorial example files and any example code from the two pdf files.
- Merge all materials onto the current master branches of the OpenJML, Specs, Solvers, and JMLAnnotations repositories
- Be sure that all tests run successfully
- Update the version number
- Push those master branches to the github repo. The push of the OpenJML current master branch (master-21) as of this writing) triggers the automated release build as a github action. The release build process includes running the release tests, requiring a successful result, but does not run the entire test suite.
- The github action creates a release candidate (a "draft" release) on the releases page of the OpenJML project
- The human in charge of the release then needs to verify that the release built and deployed successfully, edit the release notes, and then "publish" the release on GitHub.

14.8 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. The procedure is maintained on the wiki here:
https://github.com/OpenJML/OpenJML/wiki/JavaUpdate.

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*>	[§5.8] read options from a file. *This is implemented only for Java options, not* OpenJML *options*
-Akey	[§5.8] options to pass to annotation processors
--add-modules <*modulelist*>	[§5.11] see Java documentation re modules
-bootclasspath <*path*> --boot-class-path <*path*>	[§5.8] see Java documentation
-cp <*path*> -classpath <*path*> --classpath <*path*>	[§4.2, §5.8] location of input class files
-d <*directory*>	[§5.8] location of output class files
-deprecation	[§5.8] warn about use of deprecated features
--enable-preview	enables preview language features
-encoding <*encoding*>	[§5.8] character encoding used by source files
-endorseddirs <*dirs*>	[§5.8] see Java documentation
-extdirs <*dirs*>	[§5.8] see Java documentation
-g	[§5.8] generate debugging information
-h <*directory*>	location of generated header files
-? -help --help	[§5.6] output (Java and JML) help information
--help-extra	[§5.6] help about extra Java options
-implicit	[§5.8] whether or not to generate class files for implicitly referenced classes
-J<*flag*>	[§5.8] flags for the runtime system
--limit-modules <*modulelist*>	[§5.11] see Java documentation re modules
-m <*modulelist*> --module <*modulelist*>	[§5.11] see Java documentation re modules
--module-path <*path*>	[§5.11] see Java documentation re modules
--module-source-path <*path*>	[§5.11] see Java documentation re modules
--module-version <*version*>	[§5.11] see Java documentation re modules
-nowarn	[§5.9] show only errors, suppressing warning messages

Options inherited from OpenJDK (cont.) See the Java documentation for more detail	
-p *<path>*	[§5.11] like --module-path see Java documentation re modules
-parameters	see Java documentation
-proc	[§5.10] see Java documentation re annotation processing
-processor *<classes>*	[§5.10] see Java documentation re annotation processing
--processor-module-path *<path>*	[§5.10] see Java documentation re annotation processing
-processorpath *<path>* --processor-path *<path>*	[§5.10] where to find annotation processors see Java documentation re annotation processing
-profile	[§5.8] see Java documentation
--release *<release>*	[§5.7] target release for compilation
-s *<directory>*	[§5.8] location of output source files
-source *<release>* --source *<release>*	[§5.7] the Java version of source files
-sourcepath *<path>* --source-path *<path>*	[§4.2] location of source files
--system *<jdk>*	[§5.8] see Java documentation
-target *<release>* --target *<release>*	[§5.7] the Java version of the output class files
--upgrade-module-path *<path>*	[§5.11] see Java documentation re modules
-verbose	[§5.6] verbose output for Java compiler only, not OpenJML
-version --version	[§5.6] output (OpenJML) version
-Werror	[§3.4, §5.9] treat warnings as errors
-X	[§5.6] Java non-standard extensions
-Xlint	[§5.9] enable OpenJDK lint warnings

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

Options specific to JML Options listed as spelled with two initial hyphens may also be spelled with just one hyphen, with two preferred	
--arithmetic-failure *<mode>*	[§4.11.1] sets the mode for arithmetic checks: hard, soft (the default) or quiet
--check	[§5.3] typecheck only (**--command=check**)
--check-accessible -checkAccessible	[§7.3.5.1] whether to check accessible clauses (default: true)
--check-feasibility *<list>* -checkFeasibility *<list>*	[§7.2] kinds of feasibility to check (default: none)
--code-math *<mode>*	[§4.11] arithmetic mode for Java code (default: safe)
--command *<action>*	[§5.3] which action to do: `check esc rac compile parse doc`, default is `check`
--compile	[§5.3] typecheck JML but compile just the Java code (**--command=compile**)
--counterexample -ce	[§7.3.7] show a counterexample for failed static checks (default: off)
--defaults *<list>*	enables various default behaviors - *in development*
--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	[§4.12] whether to use bit-vector arithmetic (default: auto)
--esc-max-warnings *<n>* -escMaxWarnings *<n>*	[§7.1.1§7.3.6] max number of verification errors to report (default: unlimited)
--esc-warnings-path	[§7.1.1 §7.3.6] whether to find all failure paths to a failing assertion (default: off)
--exec *<file>*	[§7.3.3] file path to prover executable (default: installed executable)
--exclude *<patterns>*	[§7.3.4] paths to exclude from verification (default: no exclusions)
--extensions *<classes>*	[§11] comma-separated list of extensions classes and packages (default: no additional extensions)
--infer *<list>*	[§12.1] comma-separated list of keys designating specifications to infer if possible
--inline-function-literal	*Experimental*
-java	[§5.2] use the native OpenJDK tool, with no JML features (default: off)
-jml	[§5.2] process JML constructs (default: on)
--jmldebug	[§5.6] very verbose output (includes --progress) (**--verboseness=4**)
-jmltesting	[§5.6] changes some behavior for testing (default: false)
--jmlverbose	[§5.6] JML-specific verbose output (**--verboseness=3**)
--keys *<list>*	[§4.5] define keys for optional annotations (default: no keys)
--lang *<language>*	[§9] the JML variant to use (openjml (the default), jml)
--method *<patterns>*	[§7.3.4] methods to include in verification (default: all methods)
--nonnull-by-default -nonnullByDefault	[§5.4] type uses are not null by default (default: on)
--normal	[§5.6] only outputs diagnostics; no other progress information (**--verboseness=1**)
--nullable-by-default -nullableByDefault	[§5.4] values may be null by default (default: off)
--os-name *<name>*	[§7.3.3] Operating System name to use in selecting prover (default: (auto), or one of `macos`, `linux`, `windows`)
--parse	[§5.3] parsing only (**--command=parse**)

Options specific to JML (cont.) Options listed as spelled with two initial hyphens may also be spelled with just one hyphen, with two preferred	
--progress	[§5.6] outputs errors, warnings, progress and summary information (**--verboseness=2**)
--properties *<file>*	[§4.3.3] property file to read (no default)
--prover *<name>*	[§7.3.3] prover to use (default: z3-4.3)
--purity-check -purityCheck	[§6.3] check for purity in libraries (default: on)
--quiet	[§5.6] no output at all, just exit code (**--verboseness=0**)
--rac	[§5.3] compile runtime assertion checks (**--command=rac**)
--rac-check-assumptions -racCheckAssumptions	[§8.3.1] enables (default on) checking assume statements as if they were asserts
--rac-compile-to-java-assert -racCompileToJavaAssert	[§8.3.7] compile RAC checks using Java asserts (default: off)
--rac-java-checks -racJavaChecks	[§8.3.2] enables (default off) performing JML checking of violated Java features
--rac-missing-model-field-rep	[§8.3.9] controls behavior when model fields have no representation
--rac-precondition-entry -racPreconditionEntry	[§8.3.8] distinguishes precondition failures from external vs internal calls
--rac-show-source -racShowSource	[§8.3.6] includes source location in RAC assertion failure messages (values: `none`, `line`, `source`); default is `line`
--require-white-space	[§6.1] whether white space is required after an @ (default: false)
--show *<list>*	[§5.6] prints the details of source transformation (default: false)
--show-not-executable -showNotExecutable	[§8.3.4] warn about features not executable, in --rac operations (default: off)
--show-not-implemented -showNotImplemented	[§8.3.5] warn about features not implemented (default: off)
--show-skipped -skipped	[§7.3.4] show methods whose proofs are skipped (default: true)
--show-summary	[§7.3.9] shows a summary of numbers of methods and classes proved in one run of OpenJML (requires --progress)
--smt *filename*	[§7.3.11] where to write generated SMT files (for off-line use or inspection): default is no output
--solver-seed *<n>*	[§7.3.11] seed to pass on to the SMT solver (default: 0 - no seed)
--spec-math *<mode>*	[§4.11] arithmetic mode for specifications (default: bigint)
--specs-path *<paths>* -specspath	[§4.2] location of specs files
--split *<split-keys>*	[§9.3.6.6] splits proof of method into sections (default: no split)
-staticInitWarning	*Experimental*
--subexpressions	[§7.3.7] show subexpression detail for failed static checks (default: false)
--timeout *<seconds>*	[§7.3.11] timeout for individual prover attempts (default: no limit)
--trace	[§7.3.7] show a trace for failed static checks (default: false)
--triggers	enable SMT triggers (default: true)
-typeQuants	*Experimental*
--verboseness *<n>*	[§5.6] level of verboseness (-1=silent, 0=quiet .. 4=jmldebug) (default: 1, **--normal**)
--verify-exit *<n>*	[§7.3.11] exit code for verification failures (default: 6)
--warn *<list>*	[§5.9] 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 most 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** (§8.3.1). 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`	[§B.3] an expression uses memory locations that violate an `accessible` clause
`ArithmeticCastRange`	[§B.4] the argument for an arithmetic cast operation is out of range for the target type
`ArithmeticOperationRange`	[§B.5] the result of an arithmetic operation is out of range for its result type
`Assert`	[§B.6] an explicit assert cannot be proved valid
`Assignable`	[§B.7] an assignment or method call violates an `assignable` clause
`Callable`	[§B.9] a method call violates a `callable` clause
`CompletePreconditions`	[§B.10] the preconditions of the method are not complete
`Constraint`	[§B.11] a `constraint` clause is not proved valid as part of a method postcondition
`DisjointPreconditions`	[§B.12] the preconditions of the method are not disjoint
`ExceptionalPostcondition`	[§B.13] an exceptional postcondition (`signals` clause) is not proved valid
`ExceptionList`	[§B.14] an exception is thrown that is not in the `signals_only` exception list
`IllegalArgument`	[§B.15] a miscellaneous illegal (other than null) argument in a JML expression
`Initially`	[§B.16] an `initially` clause is not valid as part of a constructor postcondition
`Invariant`	[§B.17]
`InvariantEntrance`	[§B.18] an invariant is false on entrance to a method
`InvariantExceptionExit`	[§??] an invariant is false on exceptional exit from a method
`InvariantExit`	[§B.20] an invariant is false on normal exit from a method
`LoopDecreases`	[§B.22] the value in a loop `decreases` clause does not decrease in a loop iteration

Static warnings about assertions (cont.)	
Warning class	Description
`LoopDecreasesNonNegative`	[§B.23] the value in a loop `decreases` clause is negative at the beginning of a loop iteration
`LoopInvariant`	[§B.24] a loop invariant is not valid after the body of a loop
`LoopInvariantAfterLoop`	[§B.25] a loop invariant is not valid on exit from the loop
`LoopInvariantBeforeLoop`	[§B.26] a loop invariant is not valid before the first iteration of the loop
`NullArgument`	[§B.27] a non-null method or JML function argument may be null
`NullField`	[§B.28] as part of the postcondition of a method, a class field declared non_null
`NullFormal`	[§B.27] a non-null method or JML function argument cannot be proved to be not null
`PossiblyBadArrayAssignment`	[§B.29] assignment of a reference to an array where the reference type is not a subtype of the underlying array index type (a Java `ArrayStoreException`)
`PossiblyBadCast`	[§B.30] a reference expression cannot be proved to have the type requested in the cast
`PossiblyDivideByZero`	[§B.31] the denominator of a division operation might be 0
`PossiblyLargeShift`	[§B.32] the shift amount in a left shift operation is negative or larger or equal to the number of bits in the left-hand argument (this is not illegal in Java, but usually surprises users)
`PossiblyNegativeIndex`	[§B.33] the index of an array index operation might be negative
`PossiblyNegativeSize`	[§B.34] an array creation expression might have a negative size
`PossiblyNullAssignment`	[§B.35] a null value is being assigned to a NonNull location
`PossiblyNullDeReference`	[§B.36] an expression being dereferenced may be null
`PossiblyNullInitialization`	[§B.37] a NonNull field or variable is being initialized with a null value
`PossiblyNullReturn`	[§B.38] a null reference is returned from a method with a non-null return type
`PossiblyNullUnbox`	[§B.39] a null reference is being unboxed to a primitive
`PossiblyNullValue`	[§B.40] the value for a switch, throw, or synchronized statement is null
`PossiblyTooLargeIndex`	[§B.41] the index of an array index operation is larger or equal to the array length
`Postcondition`	[§B.42] a postcondition (`ensures` clause) is not valid
`Precondition`	[§B.43] the composite precondition of a method being called cannot be proved valid
`Reachable`	[§B.44] there is no execution path to a `reachable` statement (ESC only)
`Readable-if`	[§B.45] 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
`TerminationDecreases`	[§B.46] a `measured_by` value cannot be proved to decrease in a recursive calling chain
`TerminationNonNegative`	[§B.47] a `measured_by` value cannot be proved to stay non-negative in a recursive call-chain
`UndefinedBadCast`	[§B.30 §B.48] within a JML expression, a reference expression cannot be proved to have the type requested in the cast
`UndefinedBigintUSR`	[§B.49] within a JML expression, an unsigned right shift of a negative `\bigint` value is not well-defined

Static warnings about assertions (cont.)	
Warning class	Description
`UndefinedLemmaPrecondition`	[§B.50 §B.48] the precondition of a lemma in a `use` statement cannot be proved true
`UndefinedNegativeIndex`	[§B.33 §B.48] the index of an array index operation is negative in a JML expression
`UndefinedNegativeSize`	[§B.34 §B.48] the size of an array is negative in a JML expression
`UndefinedNullDeReference`	[§B.36 §B.48] an expression being dereferenced is null in a JML expression
`UndefinedNullUnbox`	[§B.39 §B.48] a null reference is being unboxed to a primitive in a JML expression
`UndefinedNullValue`	[§B.40 §B.48] in a JML expression, an expression in a switch, throw or syncronized expression is null
`UndefinedPrecondition`	[§B.43 §B.48] the precondition of a (pure) method being called in a JML expression does not hold
`UndefinedTooLargeIndex`	[§B.41 §B.48] the index of an array index operation is larger or equal to the array length in a JML expression
`Unreachable`	[§B.51] there is an execution path to a `unreachable` statement
`Writable-if`	[§B.52] a field is written when the writable-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
`Assume`	[§B.8] an explicit assume statement is found to be invalid
`ImplicitAssume`	[§??] reported when an implicit assumption, generated internally by OpenJML, is found to be invalid
`InvariantEntranceAssumed`	[§B.19] reported when the assumption of an invariant at the beginning of a method is false
`InvariantExitAssumed`	[§B.21] reported when the assumption of a callee's invariant in the caller after the normal or exceptional return from the callee is false
`LoopInvariantAssumption`	[§??]
`NullField`	[§B.28] a class field designated `non_null` is found to be null when assumed at the beginning of a called method
`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)

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 on your system, 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:

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

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.

These examples are available as part of the release and are tested as part of OpenJML testing.

B.3 `Accessible` warning

An `Accessible` verification warning is reported whenever a memory reference is found in the body of a method that is not contained in the method's `reads` clause.

This example source code

```
//@ @org.jmlspecs.annotation.Options("--check-accessible")
public class Demo {

  public int i, j;

  //@ reads i;
  //@ pure
  public int mok() {
    return i; // OK
  }

  //@ reads \nothing;
  public int mbad() {
    return i; // ERROR
  }

  //@ reads j;
  public int mbad2() {
    return mok(); // ERROR
  }
}
```

produces this output (when run with --**check-accessible** enabled)

```
Demo.java:14: verify: The prover cannot establish an assertion (Accessible: Demo.java:12:)
      in method mbad: i
    return i; // ERROR
              ^
Demo.java:12: verify: Associated declaration: Demo.java:14:
  //@ reads \nothing;
      ^
Demo.java:19: verify: The prover cannot establish an assertion (Accessible: Demo.java:17:)
      in method mbad2: i
    return mok(); // ERROR
              ^
Demo.java:17: verify: Associated declaration: Demo.java:19:
  //@ reads j;
      ^
4 verification failures
```

B.4 ArithmeticCastRange warning

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

```
public class Demo {

  //@ requires 0 <= i && i < 32768;
  static public void demo(int i) {
    short k = (short)i;
    byte b = (byte)i;
  }
}
```

The result of ESC is

```
Demo.java:6: verify: The prover cannot establish an assertion (ArithmeticCastRange) in
      method demo
    byte b = (byte)i;
             ^
1 verification failure
```

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.5 ArithmeticOperationRange warning

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

```
public class Demo {

  static public void demo(int i) {
    int kkk = i + i + i;
    int k = i * i;
  }

  //@ requires i >= 0 && i < 32000;
  static public void demo2(int i) {
    int kk = i + i;
    int k = i * i * i * i;
  }
}
```

The result of ESC is

```
Demo.java:5: verify: The prover cannot establish an assertion (ArithmeticOperationRange) in
     method demo: int multiply overflow
    int k = i * i;
              ^

Demo.java:4: verify: The prover cannot establish an assertion (ArithmeticOperationRange) in
     method demo: underflow in int sum
    int kkk = i + i + i;
                ^

Demo.java:4: verify: The prover cannot establish an assertion (ArithmeticOperationRange) in
     method demo: overflow in int sum
    int kkk = i + i + i;
                ^

Demo.java:4: verify: The prover cannot establish an assertion (ArithmeticOperationRange) in
     method demo: overflow in int sum
    int kkk = i + i + i;
                    ^

Demo.java:4: verify: The prover cannot establish an assertion (ArithmeticOperationRange) in
     method demo: underflow in int sum
    int kkk = i + i + i;
                    ^

Demo.java:11: verify: The prover cannot establish an assertion (ArithmeticOperationRange) in
      method demo2: int multiply overflow
    int k = i * i * i * i;
                      ^

Demo.java:11: verify: The prover cannot establish an assertion (ArithmeticOperationRange) in
      method demo2: int multiply overflow
    int k = i * i * i * i;
                  ^

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; however 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.6 `Assert` warning

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

```
public class Demo {

  static public int demo(int i) {
    if (i > 0) return 1;
    //@ assert i < 0;
    return i;
  }

  //@ requires i >= 0;
  static public int demo2(int i) {
    if (i > 0) return 1;
    //@ assert i == 0;
    return i;
  }
}
```

The result of ESC is

```
Demo.java:5: verify: The prover cannot establish an assertion (Assert) in method demo
    //@ assert i < 0;
        ^
1 verification failure
```

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.7 `Assignable` warning

An `Assignable` verification warning indicates that a memory location is being written to that is not listed in the frame condition of the method's specification.

For example, this example

```
public class Demo {
  public int f;

  //@ assigns \nothing;
  public void demo(int i) {
    f = i; // ERROR - f is not in assigns clause
  }
}
```

produces this output

```
Demo.java:6: verify: The prover cannot establish an assertion (Assignable: Demo.java:4:) in
     method demo: f
    f = i; // ERROR - f is not in assigns clause
```

```
      ^
Demo.java:4: verify: Associated declaration: Demo.java:6:
  //@ assigns \nothing;
      ^
2 verification failures
```

B.8 `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

```
public class Demo {

  static public int demo(int i) {
    if (i > 0) return 1;
    //@ assume i < 0;
    return i;
  }
  @org.jmlspecs.annotation.SkipEsc
  @org.jmlspecs.annotation.SkipRac
  public static void main(String ... args) {
     int i = args.length == 0 ? 0 : Integer.parseInt(args[0]);
     demo(i);
  }
}
```

compiled with

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

and run with

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

results in

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

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

B.9 `Callable` warning

The `callable` clause in a method specification lists the methods that the method may call directly or may call indirectly through any callee methods. A `Callable` verification warning is issued if a method calls an unlisted method. Because the default `callable` clause is `callable \everything;`, this verification warning only occurs when there are explicit `callable` clauses. See the discussion of the `callable` clause in the JML Reference Manual for more detail.

Here is an example:

```
public class Demo {
  //@ callable \nothing;
  public static void test1() {}
  //@ callable \nothing;
  public static void test2() {}
  //@ callable test2;
  public static void main(String ... args ) { test1(); }
}
```

The result of ESC on this example is

```
Demo.java:7: verify: The prover cannot establish an assertion (Callable: Demo.java:6:) in
     method main: Demo.test1() is not callable
  public static void main(String ... args ) { test1(); }
                                                   ^
Demo.java:6: verify: Associated declaration: Demo.java:7:
  //@ callable test2;
      ^
2 verification failures
```

B.10 `CompletePreconditions` warning

A `CompletePreconditions` failure is issued when the preconditions of a method are found to violate a `behaviors local_complete` or `behaviors complete` clause in the method specifications (§9.2).

Here are two examples. This code

```
public class Demo {

  //@   requires i < 0;
  //@ also
  //@   requires i > 0;
  //@ behaviors complete;
  public void m(int i) {}

}
```

fails with `openjml --esc Demo.java`, giving this output

```
Demo.java:6: verify: The prover cannot establish an assertion (CompletePreconditions) in
     method m
  //@ behaviors complete;
      ^
1 verification failure
```

because the preconditions of the two specification cases together do not cover all possibilities for the input state (here just the value of i).

A second example shows the use of `local_complete`.

```
class Parent {
  //@ requires i == 0;
  public void m(int i) {}
}

public class Demo extends Parent {

  //@ also
  //@   requires i < 0;
  //@ also
  //@   requires i > 0;
  //@ behaviors complete;  // OK
  //@ behaviors local_complete; // ERROR
  @Override
  public void m(int i) {}

}
```

produces

```
Demo.java:13: verify: The prover cannot establish an assertion (CompletePreconditions) in
     method m
  //@ behaviors local_complete; // ERROR
      ^
1 verification failure
```

B.11 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:

```
public class Demo {

  private /*@ spec_public */ int count;

  //@ public constraint count > \old(count);

  //@ requires count < Integer.MAX_VALUE;
  //@ assignable count;
  public void increment() {
    count++;
  }

  //@ assignable \nothing;
  //@ ensures \result == count;
  public int count() {
    return count;
  }
}
```

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:

```
Demo.java:16: verify: The prover cannot establish an assertion (Constraint: Demo.java:5:) in
      method count
    return count;
    ^
Demo.java:5: verify: Associated declaration: Demo.java:16:
  //@ public constraint count > \old(count);
             ^
2 verification failures
```

B.12 DisjointPreconditions warning

A `DisjointPreconditions` failure is issued when the preconditions of a method are found to violate the `behaviors local_disjoint` or `behaviors disjoint` clauses in the method specifications (§9.2).

Here are two examples. This code

```
public class Demo {

  //@   requires i <= 0;
  //@ also
  //@   requires i >= 0;
  //@ behaviors disjoint;
  public void m(int i) {}

}
```

fails with `openjml --esc Demo.java`, giving this output

```
Demo.java:5: verify: The prover cannot establish an assertion (DisjointPreconditions: Demo.
     java:3:) in method m
  //@   requires i >= 0;
        ^
Demo.java:3: verify: Associated declaration: Demo.java:5:
  //@   requires i <= 0;
        ^
2 verification failures
```

because the preconditions of the two specification cases overlap: the value of 0 for i satisfies both.

A second example shows the use of `local\_disjoint`.

```
class Parent {
  //@ requires i == 0;
  public void m(int i) {}
}

public class Demo extends Parent {

  //@ also
  //@   requires i < 0;
  //@ also
  //@   requires i >= 0;
  //@ behaviors disjoint;  // ERROR
  //@ behaviors local_disjoint; // OK
  @Override
  public void m(int i) {}

}

class Child extends Parent {

  //@ also
  //@   requires i <= 0;
  //@ also
  //@   requires i >= 0;
  //@ behaviors local_disjoint; // ERROR
  @Override
  public void m(int i) {}

}
```

produces

```
Demo.java:11: verify: The prover cannot establish an assertion (DisjointPreconditions: Demo.
      java:2:) in method m
  //@   requires i >= 0;
        ^
Demo.java:2: verify: Associated declaration: Demo.java:11:
  //@ requires i == 0;
      ^
Demo.java:24: verify: The prover cannot establish an assertion (DisjointPreconditions: Demo.
      java:22:) in method m
  //@   requires i >= 0;
        ^
Demo.java:22: verify: Associated declaration: Demo.java:24:
  //@   requires i <= 0;
        ^
4 verification failures
```

B.13 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

$$(e\ \texttt{instanceof}\ \mathit{Exc}) \rightarrow \mathit{expr}$$

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.

```
public class Demo {

  static public int field;

  //@ ensures  field == i;
  //@ signals (Exception e) field == i;
  static public void demo(int i) {
    init();
    field = i;
  }

  /*@ pure */ static void init() {
  }
}
```

Applying ESC to this example indeed produces an ExceptionalPostcondition warning:

```
Demo.java:8: verify: The prover cannot establish an assertion (ExceptionalPostcondition:
      Demo.java:6:) in method demo
    init();
        ^
Demo.java:6: verify: Associated declaration: Demo.java:8:
  //@ signals (Exception e) field == i;
      ^
2 verification failures
```

B.14 ExceptionList warning

An ExceptionList warning is issued when a method might terminate with an exception that is not included in a signals_only clause in the method's specification. The warning can occur in both RAC and ESC testing.

Here is an example:

```
public class Demo {
  //@ signals_only NullPointerException;
  public static void test() throws Exception {
    throw new java.io.FileNotFoundException();
  }
  public static void main(String ... args) throws Exception { test(); }
}
```

The result of ESC on this example is

```
Demo.java:4: verify: The prover cannot establish an assertion (ExceptionList: Demo.java:2:)
      in method test
    throw new java.io.FileNotFoundException();
    ^
Demo.java:2: verify: Associated declaration: Demo.java:4:
  //@ signals_only NullPointerException;
      ^
2 verification failures
```

B.15 IllegalArgument warning

An IllegalArgument warning is issued when a JML expression has an illegal argument (other than null). The warning can occur in both RAC and ESC testing.

Here is an example:

```
public class Demo {
    public static void test() {
        //@ check \elemtype(\type(Object)) == \type(boolean);
    }
}
```

The result of ESC on this example is

```
Demo.java:3: verify: The prover cannot establish an assertion (IllegalArgument) in method test
        //@ check \elemtype(\type(Object)) == \type(boolean);
                           ^
1 verification failure
```

B.16 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:

```
public class Demo {

  public int count;

  //@ public initially count > 0;

}
```

The result of ESC on this example is

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

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.17 Invariant warning

Needs example

B.18 InvariantEntrance warning

An `InvariantEntrance` warning is issued if a callee invariant is not true when a calling method is about to call a callee method. The warning can occur in both RAC and ESC testing.

Here is an example:

```
public class Demo {
  public static void main(String... args) {
    i = 1;
    test();
    i = 0;
  }
  public static void test() {
    i = 0;
  }
  public static int i = 0;
  //@ public static invariant i == 0;
}
```

The result of ESC on this example is

```
Demo.java:4: verify: The prover cannot establish an assertion (InvariantEntrance: Demo.java:11:) in method main: (Caller: Demo.main(String...), Callee: Demo.test())
    test();
        ^
Demo.java:11: verify: Associated declaration: Demo.java:4:
  //@ public static invariant i == 0;
                    ^
2 verification failures
```

B.19 InvariantEntranceAssumed warning

At the beginning of a method body, all relevant invariants are assumed, along with the method's preconditions. If any assumed invariant is found to be false at runtime, this `InvariantEntranceAssumed` warning is issued. The warning can only occur in RAC and only with **--rac-check-assumptions**.

Here is an example:

```
public class Demo {
  public static void main(String... args) {
    i = 1;
    test();
  }
  public static void test() {
    i = 0;
  }
  public static int i = 0;
  //@ public static invariant i == 0;
}
```

The result of `openjml --rac --rac-check-assumptions Demo.java && openjml-java Demo` on this example is

```
Demo.java:4: verify: JML invariant is false on entering method (Caller: Demo.main(java.lang.String...), Callee: Demo.test())
    test();
        ^
Demo.java:10: verify: Associated declaration: Demo.java:4:
  //@ public static invariant i == 0;
                    ^
Demo.java:6: verify: JML assumed invariant is false on entering method Demo.test()
  public static void test() {
                     ^
Demo.java:10: verify: Associated declaration: Demo.java:6:
  //@ public static invariant i == 0;
                    ^
```

Note that the invariant is checked by the caller before calling the callee and it is assumed by the callee, but also checked because of the --**rac-check-assumptions** option. Accordingly the example shows both verification failure messages. However, if the caller is not compiled with RAC, then only the callee failure would be reported.

B.20 `InvariantExit` warning

An `InvariantExit` warning is issued if an invariant of a class is no longer true when a method returns. The warning can occur in both RAC and ESC testing and whether the callee method returns normally or with an exception.

Here is an example:

```
public class Demo {
  public int i = 0;
  //@ public invariant i >= 0;
  public void test() {
    i = -1;
  }
}
```

The result of ESC on this example is

```
Demo.java:4: verify: The prover cannot establish an assertion (InvariantExit: Demo.java:3:) in method test
  public void test() {
              ^
Demo.java:3: verify: Associated declaration: Demo.java:4:
  //@ public invariant i >= 0;
             ^
2 verification failures
```

B.21 `InvariantExitAssumed` warning

After the return from calling a method, all relevant callee invariants are assumed by the caller, along with the method's postconditions. If any assumed invariant is found to be false at runtime, this `InvariantExitAssumed` warning is issued. The invariants are checked by the caller for both the normal and exceptional returns from the callee. The warning can only occur in RAC and only with --**rac-check-assumptions**.

Here is an example:

```
public class Demo {
  public static void main(String... args) {
    test();
    i = 0;
  }
  public static void test() {
    i = 1;
  }
  public static int i = 0;
  //@ public static invariant i == 0;
}
```

The result of `openjml --rac --rac-check-assumptions Demo.java && openjml-java Demo` on this example is

```
Demo.java:6: verify: JML invariant is false on leaving method Demo.test()
  public static void test() {
                     ^
Demo.java:10: verify: Associated declaration: Demo.java:6:
  //@ public static invariant i == 0;
                    ^
Demo.java:3: verify: JML assumed invariant is false on leaving method Demo.test(), returning
      to Demo.main(java.lang.String...)
    test();
        ^
Demo.java:10: verify: Associated declaration: Demo.java:3:
  //@ public static invariant i == 0;
                    ^
```

Note that the invariant is checked by the callee before returning to the caller and it is subsequently assumed by the caller, but in RAC with the **--rac-check-assumptions** option that assumption is checked. Accordingly the example shows both verification failure messages. However, if the callee is not compiled with RAC, then only the caller failure would be reported.

B.22 `LoopDecreases` warning

A `LoopDecreases` verification failure is issued if OpenJML cannot prove that some expression (the *variant*) decreases on each iteration of a loop; that is, its value evaluated at the end of the loop is less than the value evaluated at the beginning of the loop. The expression evaluated is given in a `loop_decreases` clause.

Applying ESC to this example

```
public class Demo {
  public void m() {
    //@ loop_invariant 0 <= i <= 10;
    //@ loop_decreases i; // Does not decrease
    for (int i=0; i<10; i++) {}

    //@ loop_invariant 0 <= k <= 10;
    //@ loop_decreases 10-k; // OK - decreases
    for (int k=0; k<10; k++) {}
  }
}
```

results in this output:

```
Demo.java:4: verify: The prover cannot establish an assertion (LoopDecreases) in method m
    //@ loop_decreases i; // Does not decrease
        ^
1 verification failure
```

B.23 LoopDecreasesNonNegative warning

As well as decreasing on each loop iteration, the expression in a `loop_decreases` clause must also never be negative at the beginning of an execution of the loop body. If cannot be proved to be non-negative for any value of the loop index that satisfies the loop invariants and the loop condition, then a `LoopDecreasesNonNegative` verification failure is issued.

Applying ESC to this example

```
public class Demo {
  public void m() {
    //@ loop_invariant 0 <= j <= 10;
    //@ loop_decreases 9-j; // OK - is -1 only on loop exit
    for (int j=0; j<10; j++) {}

    //@ loop_invariant 0 <= k <= 10;
    //@ loop_decreases 10-k; // ok - best style
    for (int k=0; k<10; k++) {}

    //@ loop_invariant 0 <= i <= 10;
    //@ loop_decreases -i; // Becomes negative
    for (int i=0; i<10; i++) {}
  }
}
```

results in this output:

```
Demo.java:12: verify: The prover cannot establish an assertion (LoopDecreasesNonNegative) in
      method m
    //@ loop_decreases -i; // Becomes negative
        ^
1 verification failure
```

B.24 LoopInvariant warning

Each expression given in a `loop_invariant` clause must be maintained by the loop body; that is, assuming the expression is true at the beginning of the loop and the loop guard is also true, then the invariant must be true at the end of the loop. If this property cannot be proved, a `LoopInvariant` verification failure is reported.

Applying ESC to this example

```
public class Demo {
  public void m() {
    int i = 0;
    //@ loop_invariant 0 <= i < 10; // Not valid after last
```

```
                                 // iteration, when i is 10
    //@ loop_decreases 10-i;
    while (i<10) i++;
  }
}
```

results in this output:

```
Demo.java:4: verify: The prover cannot establish an assertion (LoopInvariant) in method m
    //@ loop_invariant 0 <= i < 10; // Not valid after last
        ^
1 verification failure
```

B.25 LoopInvariantAfterLoop warning

Each expression given in a `loop_invariant` clause must be true when the loop terminates, if it terminates normally and not by a `break` statement. If this condition cannot be proved, a `LoopInvariantAfterLoop` verification failure is reported. Recall though that the loop invariant must be true at the end of the just completed loop iteration, so a `LoopInvariantAfterLoop` warning can only occur if evaluating the loop condition causes some change in state.

Applying ESC to this example

```
public class Demo {
  public static void main(String ... args) {
    int i = 0;
    //@ loop_invariant 0 <= i <= 10;
    //@ loop_decreases 10 - i;
    while ( i++ < 10 ) {}
    System.out.println("END " + i);
  }
}
```

results in this output:

```
Demo.java:4: verify: The prover cannot establish an assertion (LoopInvariantAfterLoop) in
     method main
    //@ loop_invariant 0 <= i <= 10;
        ^
1 verification failure
```

B.26 LoopInvariantBeforeLoop warning

Each expression given in a `loop_invariant` clause must be proved to be true before a loop begins (but after the initialization in a `for` loop). If this property cannot be proved, a `LoopInvariantBeforeLoop` verification failure is reported.

Applying ESC to this example

```
public class Demo {
  public void m() {
    int i = -1;
    //@ loop_invariant 0 <= i <= 10;
    //@ loop_decreases 10-i;
    while (i<10) i++;
  }
}
```

results in this output:

```
Demo.java:4: verify: The prover cannot establish an assertion (LoopInvariantBeforeLoop) in
     method m
    //@ loop_invariant 0 <= i <= 10;
        ^
1 verification failure
```

B.27 NullArgument warning, NullFormal warning

If the actual argument of a method or JML function whose formal argument is declared non-null or whose semantics specify that the argument must be non-null, then if the actual argument cannot be proved to be non-null, a NullArgument or NullFormal verification warning is emitted.

These two warning differ only in that NullFormal includes some additional information and is typically used to report null arguments of method calls.

A NullField verification warning is issued if OpenJML cannot determine that such a field is non-null when a method terminates.

This warning may occur in both ESC and RAC testing.

Here is an example:

```
public class Demo {

  public static void test(/*@ nullable */ Object o) {
    //@ ghost \TYPE t = \typeof(o);
  }

  public static void tp(/*@ nullable */ Object o) { m(o); }

  public static void m(/*@ non_null */ Object o) { }

}
```

The result of ESC on this example is

```
Demo.java:4: verify: The prover cannot establish an assertion (NullArgument) in method test
    //@ ghost \TYPE t = \typeof(o);
                                ^
Demo.java:7: verify: The prover cannot establish an assertion (NullFormal: Demo.java:9:) in
     method tp: o in m(@NonNull Object)
  public static void tp(/*@ nullable */ Object o) { m(o); }
                                                      ^
Demo.java:9: verify: Associated declaration: Demo.java:7:
  public static void m(/*@ non_null */ Object o) { }
                           ^
3 verification failures
```

B.28 NullField warning

If a class has fields whose types are implicitly or explicitly non-null, then the class has implicit invariants that such fields have non-null values. Consequently all methods have an implicit precondition that the fields are non-null and all methods and constructors have an implicit postcondition that such fields are non-null.

A NullField verification warning is issued if OpenJML cannot determine that such a field is non-null when a method terminates. This warning may occur in both ESC and RAC testing. The warning may also occur in RAC with **--rac-check-assumptions**, if the assumption that a field is non-null at the beginning of a callee method is false; such a situation would also result in an assertion violation because the caller will check that the field is non-null, but only if the caller is also compiled for RAC checking.

Here is an example:

```
public class Demo {
  Object o;
  public Demo() {}
}
```

The result of ESC on this example is

```
Demo.java:2: verify: The prover cannot establish an assertion (NullField) in method Demo
  Object o;
         ^
1 verification failure
```

B.29 PossiblyBadArrayAssignment warning

A PossiblyBadArrayAssignment warning is issued when a value is assigned to an array element that does not match the underlying array type. A Java program will issue an ArrayStoreException for this problem; it is a type error that can only be detected by Java at run-time. The OpenJML warning is only issued in ESC verification. When a program is compiled with RAC, a Java exception is thrown if this error is encountered.

Here is an example:

```
public class Demo {
  public static void test() {
    Object[] o = new Integer[10];
    o[0] = "abc";
  }
  public static void main(String ... args) { test(); }
}
```

The result of ESC on this example is

```
Demo.java:4: verify: The prover cannot establish an assertion (PossiblyBadArrayAssignment)
      in method test
    o[0] = "abc";
         ^
1 verification failure
```

B.30 PossiblyBadCast warning

A PossiblyBadCast warning is issued when a cast is attempted of a reference value to a type that does not match the dynamic type of the value. For example, a variable typed as Object but holding a String cannot be cast to an Integer. Attempting to do so in Java causes a ClassCastException.

This verification warning is issued in OpenJML RAC testing if --rac-java-checks is used and by ESC testing.

Here is an example:

```
public class Demo {
  public static void main (String ... args) {
    Object o = "abc";
    Integer i = (Integer)o;
  }
}
```

The result of ESC on this example is

```
Demo.java:4: verify: The prover cannot establish an assertion (PossiblyBadCast) in method main: a Object cannot be proved to be a Integer
    Integer i = (Integer)o;
                ^
1 verification failure
```

B.31 PossiblyDivideByZero warning

The PossiblyDivideByZero verification failure is issued when an arithmetic divide operation has a denominator that cannot be proved to be non-zero.

This example shows such a situation (method mbad) and, in method mok how the possibility can be guarded against.

```
public class Demo {
  //@ requires i != 0;
  public void mok(int i) {
    int k = 1/i;
  }
  public void mbad(int i) {
    int k = 1/i;
  }
}
```

The result of ESC is

```
Demo.java:7: verify: The prover cannot establish an assertion (PossiblyDivideByZero) in method mbad
    int k = 1/i;
             ^
1 verification failure
```

B.32 PossiblyLargeShift warning

Java defines left and right bit-shift operations on int and long values. The number of bits shifted is given by an integral value. However the number of bits shifted is restricted to at most 31 for `int` values and 63 for `long` values: the actual argument is restricted to this range by a bit-and operation with either 31 or 63.

This restriction can be a surprise for the user and can be a silent bug in Java. For example, if an `int` value is inadvertently shifted by `-1`, it will actually be shifted by `31` bits. So, just as for operations that might overflow, OpenJML gives a warning if the right-hand operand of a shift operation is not actually in the expected range of `0..31` or `0..63`. This warning is given for operations in either Java code or JML expressions but only when the arithmetic mode is `safe-math`; the warning is given in both ESC and RAC checking.

Hers is an example:

```
public class Demo {
  public static void main(String ... args) {
    test(100);
  }
  public static int test(int i) {
    return 1 << i;
  }
}
```

The result of ESC is

```
Demo.java:6: verify: The prover cannot establish an assertion (PossiblyLargeShift) in method
      test
    return 1 << i;
             ^
1 verification failure
```

B.33 `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: verify: The prover cannot establish an assertion (PossiblyNegativeIndex) in
     method demo
    return arr[i];
              ^
1 verification failure
```

B.34 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 runtime 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

```
public class Demo {

  static public int[] demo(int i) {
    return new int[i];
  }

  //@ requires i >= 0;
  static public int[] demo2(int i) {
    return new int[i];
  }


}
```

results in this output:

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

B.35 PossiblyNullAssignment warning

A Java or ghost variable that has a non-null type may not be assigned a null value. An attempt to do so results in a PossiblyNullAssignment verification warning, both in ESC and RAC.

Applying ESC to this example

```
public class Demo {
  static Integer i;
  public static void main(String ... args) { i = null; }
}
```

results in this output:

```
Demo.java:3: verify: The prover cannot establish an assertion (PossiblyNullAssignment) in
     method main
  public static void main(String ... args) { i = null; }
                                               ^
1 verification failure
```

B.36 PossiblyNullDereference warning

Java does not allow dereferencing (extracting a field value of an object with the dot operator) a null reference value. An attempt to do so results in throwing a NullPointerException. In ESC OpenJML attempts to prove that the object value of a dot-selection operation is non-null, and issues a PossiblyNullDereference verification warning if it cannot. In RAC, using --rac-java-checks, OpenJML will issue a similar warning a run-time; OpenJML will also go on to emit the Java exception.

Applying ESC to this example

```
public class Demo {
  int i;
  public static int test(/*@ nullable */ Demo d) {
    return d.i;
  }
  public static void main(String ... args) { test(null); }
}
```

results in this output:

```
Demo.java:4: verify: The prover cannot establish an assertion (PossiblyNullDeReference) in
     method test
    return d.i;
            ^
1 verification failure
```

B.37 PossiblyNullInitialization warning

A Java or ghost variable that has a non-null type may not be initialized with a null value in its declaration; this includes the implicit initialization of a field that has no explicit initializer in its declaration. An attempt to do so results in a PossiblyNullInitialization verification warning, both in ESC and RAC.

Applying ESC to this example

```
public class Demo {
  public static void main(String ... args) { Integer i = null; }
}
```

results in this output:

```
Demo.java:2: verify: The prover cannot establish an assertion (PossiblyNullInitialization)
     in method main: i
  public static void main(String ... args) { Integer i = null; }
                                                     ^
1 verification failure
```

B.38 PossiblyNullReturn warning

A method that returns a value with a type that is implicitly or exp[licitly NonNull has an implicit postcondition: if the method returns normally, the return value is non-null.

When running ESC, if OpenJML cannot prove that the return value is non-null a PossiblyNullReturn verification warning is issued.

Furthermore, if the method being verified calls a callee method and the callee method is specified to have a non-null return value, then (along with other postconditions) there is an assumption that this implicit postcondition is true after the (normal) return of the callee within the body of the method begin verified. Therefore, if RAC is run with --rac-check-assumptions and the callee method returns a null value, in violation of its implicit postcondition, then OpenJML RAC will emit a PossiblyNullReturn verification warning.

Applying ESC to this example

```
public class Demo {
  public static Integer test1() {
    return null;
  }
  public static void test2() {
    test1();
    //@ reachable
  }
  public static void main(String ... args) { test2(); }
}
```

results in this output:

```
Demo.java:2: verify: The prover cannot establish an assertion (PossiblyNullReturn: Demo.java
    :2:) in method test1
  public static Integer test1() {
                ^
Demo.java:2: verify: Associated declaration: Demo.java:2:
  public static Integer test1() {
                        ^
Demo.java:3: verify: Associated method exit
    return null;
    ^
3 verification failures
```

Applying RAC, using --rac-check-assumptions to the same example yields

```
Demo.java:2: verify: JML null return value from method test1
  public static Integer test1() {
                ^
Demo.java:2: verify: Associated declaration: Demo.java:2:
  public static Integer test1() {
                        ^
Demo.java:6: verify: JML null return value from method test1(), checked in caller test2()
    test1();
         ^
Demo.java:2: verify: Associated declaration: Demo.java:6:
  public static Integer test1() {
                        ^
```

B.39 PossiblyNullUnbox warning

In Java, primitive types like int have a corresponding boxed reference type, such as Integer. Java allows explicit or implicit conversion of a boxed value to its corresponding primitive value. However, the boxed value must always be non-null; otherwise a

NullPointerException will be thrown. An example of a location where implicit unboxing can occur is the value in a switch statement.

OpenJML, using ESC, will attempt to prove that the source reference value of an unboxing operation is non-null; OpenJML will issue a PossiblyNullUnbox verification warning if it cannot prove non-nullness. RAC produces a corresponding warning if run using --rac-java-checks; RAC will also emit a subsequent NullPointerException.

Applying ESC to this example

```
public class Demo {
  public static void main(String ... args) { /*@ nullable */ Integer i = null; int k = (int)i
      ; }
}
```

results in this output:

```
Demo.java:2: verify: The prover cannot establish an assertion (PossiblyNullUnbox) in method
     main
  public static void main(String ... args) { /*@ nullable */ Integer i = null; int k = (int)
     i; }

     ^
1 verification failure
```

B.40 PossiblyNullValue warning

A number of locations in Java statements and expressions prohibit using null reference values, besides dereferencing and unboxing. Examples are the value in a throw statement and the value in a synchronized statement. Java will throw a NullPointerException if such an expression is found to be null.

OpenJML, using ESC, will attempt to prove that the source reference value in these cases is non-null; OpenJML will issue a PossiblyNullValue verification warning if it cannot prove non-nullness. RAC produces a corresponding warning if run using --rac-java-checks; RAC will also emit a subsequent NullPointerException.

Applying ESC to this example

```
public class Demo {
  public static void main(String ... args) {
    /*@ nullable */ RuntimeException e = null;
    if (args.length == 0) synchronized (e) {}
    throw e;
  }
}
```

results in this output:

```
Demo.java:5: verify: The prover cannot establish an assertion (PossiblyNullValue) in method
     main
    throw e;
          ^
Demo.java:4: verify: The prover cannot establish an assertion (PossiblyNullValue) in method
     main
    if (args.length == 0) synchronized (e) {}
                                       ^
2 verification failures
```

B.41 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

```
public class Demo {

  public static int[] arr;

  //@ requires 0 <= i;
  static public int demo(int i) {
    return arr[i];
  }

  //@ requires 0 <= i && i < arr.length ;
  static public int demo2(int i) {
    return arr[i];
  }

}
```

results in this output:

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

1 verification failure
```

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

B.42 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 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.

```
public class Demo {

  //@ ensures \result >= 0;
  static int mm(int i) {
    return i;
  }

  public static void main(String ... args) { mm(-1); }
}
```

The result of ESC is

```
Demo.java:5: verify: The prover cannot establish an assertion (Postcondition: Demo.java:3:)
      in method mm
    return i;
    ^
Demo.java:3: verify: Associated declaration: Demo.java:5:
  //@ ensures \result >= 0;
      ^
2 verification failures
```

B.43 `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.

```
public class Demo {

  static public void demo(int i) {
    mm(i);
  }

  //@ requires i > 0;
  //@ ensures \result == 1;
  //@ also
  //@ requires i < 0;
  //@ ensures \result == -1;
  //@ pure
  static int mm(int i) {
    if (i > 0) return 1;
    if (i < 0) return -1;
    return i;
  }
}
```

The result of ESC is

```
Demo.java:4: verify: The prover cannot establish an assertion (Precondition: Demo.java:13:)
      in method demo
    mm(i);
      ^
Demo.java:13: verify: Associated declaration: Demo.java:4:
  static int mm(int i) {
             ^
Demo.java:7: verify: Precondition conjunct is false: i > 0
  //@ requires i > 0;
                 ^
Demo.java:10: verify: Precondition conjunct is false: i < 0
  //@ requires i < 0;
                 ^
4 verification failures
```

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.

B.44 Reachable warning

The `reachable` specification statement in a method body specifies that the program location containing the statement is not dead code; that is, there is some execution path satisfying the method's preconditions that passes through this `reachable` statement.

Feasibility checking such as this is a separate kind of ESC test. It is described in more detail in §7.2.

Here is an example:

```
public class Demo {
  //@ requires b;
  public void test(boolean b) {
    if (b) {
    } else {
      //@ reachable
    }
  }
}
```

The result of ESC on this example, using
`openjml --esc --check-feasibility=reachable Demo.java`, is

```
Demo.java:6: verify: There is no feasible path to program point at reachable statement in method Demo.test(boolean)
      //@ reachable
          ^
1 verification failure
```

B.45 Readable-if warning

A `Readable-if` warning is issued if, at the point at which the value of some field is being read, the predicate in a `readable` clause for that field cannot be proved to be true. The warning can occur in both RAC and ESC testing.

Here is an example:

```
public class Demo {
  int i;
  int j;
  //@ readable i if j > 0;
  public int test() {
    return i;
  }
  public static void main(String ... args) { new Demo().test(); }
}
```

The result of ESC on this example is

```
Demo.java:6: verify: The prover cannot establish an assertion (Readable-if: Demo.java:4:) in
      method test: Demo.i
    return i;
           ^
Demo.java:4: verify: Associated declaration: Demo.java:6:
  //@ readable i if j > 0;
      ^
2 verification failures
```

B.46 TerminationDecreases warning

This warning relates to proving termination of recursive call stacks. Specifically, to prove termination, a recursive method must have a measured_by specification clause and the \bigint value of the expression in that clause must decrease on each call of the method in the call stack. This condition is checked in both ESC and RAC.

Here is an example:

```
public class Demo {
  //@ requires i > 0;
  //@ measured_by i;
  public int test(int i) {
    return i <= 0 ? 0 : test(i);
  }
  public static void main(String ... args) { new Demo().test(10); }
}
```

The result of ESC on this example is

```
Demo.java:3: verify: The prover cannot establish an assertion (TerminationDecreases: Demo.
     java:5:) in method test
  //@ measured_by i;
                  ^
Demo.java:5: verify: Associated declaration: Demo.java:3:
    return i <= 0 ? 0 : test(i);
                            ^
2 verification failures
```

B.47 TerminationNonNegative warning

This warning relates to proving termination of recursive call stacks. Specifically, to prove termination, a recursive method must have a measured_by specification clause and the \bigint value of the expression in that clause must decrease on each call of the method in the call stack but must never be negative at the beginning of a method call. This condition is checked in both ESC and RAC.

Here is an example:

```
public class Demo {
  //@ requires i >= 0;
  //@ measured_by i-5;
  public int test(int i) {
    return i <= 0 ? 0 : test(i-1);
  }
  public static void main(String ... args) { new Demo().test(10); }
}
```

The result of ESC on this example is

```
Demo.java:3: verify: The prover cannot establish an assertion (TerminationNonNegative: Demo.
     java:5:) in method test
  //@ measured_by i-5;
                   ^
Demo.java:5: verify: Associated declaration: Demo.java:3:
    return i <= 0 ? 0 : test(i-1);
                            ^
2 verification failures
```

B.48 Undefined... warning

The other warnings whose names begin with 'Undefined' have the same cause as the corresponding warnings whose names begin with 'Possibly'. The 'Possibly' warnings occur on Java code, whereas the 'Undefined' warnings occur on code in JML annotations. The presence of the Undefined warning in a JML expression means that the expression is *undefined*, neither true nor false.

B.49 UndefinedBigintUSR warning

JML expressions may include operations on mathematical (unbounded) integers, that is on `\bigint` values. Bit operations such as signed left shift (<<) and signed right shift (>>) are well-defined on `\bigint` values. However, unsigned right shift (>>>) is well-defined only if the argument is non-negative; it is not well-defined on negative values. A `UndefinedBigintUSR` warning is issued if OpenJML cannot prove that if the left argument of an unsigned right shift operation has `\bigint` type, then the value of that argument is non-negative.

Here is an example:

```
public class Demo {
  public static void main(String ... args) {
    //@ set test(-1);
  }
  //@ model public static \bigint test(\bigint i) { return i >>> 1; }
}
```

The result of ESC on this example is

```
Demo.java:5: verify: The prover cannot establish an assertion (UndefinedBigintUSR) in method
      test
  //@ model public static \bigint test(\bigint i) { return i >>> 1; }
                                                             ^
1 verification failure
```

B.50 UndefinedLemmaPrecondition warning

OpenJML allows model methods to be used as lemmas (cf. §9.3.8). Lemmas are used within JML code. At the program location where a lemma is used, any preconditions of the lemma must be true. If OpenJML cannot prove that the preconditions are true, it issues a `UndefinedLemmaPrecondition` warning.

Here is an example:

```
public class Demo {
  //@ public normal_behavior
  //@   requires p >= 0;
  //@   ensures (p&1) ==  p%2;
  //@ spec_pure
  public void mod2lemma(int p) {}

  //@ requires k <= Integer.MAX_VALUE/2 && k >= -1; // The -1 just so the counterexample is always the same
  public void m(int k) {
  //@ show k;
  k = 2*k;
  //@ use mod2lemma( (k+1) );
  boolean b = ((k+1)&1) == 1; // Error expected when k is -1
  //@ assert b;
  }
  public static void main(String ... args) { (new Demo()).m(-1); }
}
```

The result of ESC on this example is

```
Demo.java:10: verify: Show statement expression k has value ( - 1 )
  //@ show k;
           ^
Demo.java:12: verify: The prover cannot establish an assertion (UndefinedLemmaPrecondition) in method m
  //@ use mod2lemma( (k+1) );
      ^
2 verification failures
```

B.51 Unreachable warning

The `unreachable` specification statement in a method body specifies that the program location containing the statement is dead code; that is, it cannot be reached by any execution path satisfying the method's preconditions. If this cannot be proved to be the case, then an `Unreachable` verification warning is issued. This warning may occur in both ESC and RAC checking.

Here is an example:

```
public class Demo {
  public static void main(String ... args) {
    if (args.length > 0) {
      //@ unreachable
    }
  }
}
```

The result of ESC on this example is

```
Demo.java:4: verify: The prover cannot establish an assertion (Unreachable) in method main
      //@ unreachable
          ^
1 verification failure
```

B.52 Writable-if warning

A Writable-if warning is issued if, at the point at which some field is being assigned to, the predicate in a writable clause for that field cannot be proved to be true. The warning can occur in both RAC and ESC testing.

Here is an example:

```
public class Demo {
  public int i;
  public int j;
  //@ writable i if j > 0;
  public void test() {
    j = 0;
    i = 1;
  }
  public static void main(String ... args) { new Demo().test(); }
}
```

The result of ESC on this example is

```
Demo.java:7: verify: The prover cannot establish an assertion (Writable-if: Demo.java:4:) in
      method test: Demo.i
    i = 1;
    ^
Demo.java:4: verify: Associated declaration: Demo.java:7:
  //@ writable i if j > 0;
      ^
2 verification failures
```

Check that all warnings are included; finish them all. Check that they match the examples being tested.

Commented-out material to do or to delete

Bibliography

[1] ANSI-C Specification Language. https://github.com/acsl-language/acsl/. 1, 5, 76

[2] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. Deductive software verification – The KeY Book. In *Lecture Notes in Computer Science*, 2016. iii

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

[4] 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. 5

[5] 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, 5

[6] 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

[7] 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. iv

[8] David R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In Mihaela Bobaru, Klaus Havelund, GerardJ. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 472–479. Springer Berlin Heidelberg, 2011. i

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

[10] David R. Cok, 2018. http://www.openjml.org. i

[11] David R. Cok. JML and OpenJML for Java 16. In *Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs*, FTfJP 2021, pages 65–67, New York, NY, USA, 2021. Association for Computing Machinery. i

[12] 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. i, iii, iv, 5

[13] 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, 63

[14] David R. Cok and Serdar Tasiran. Practical methods for reasoning about Java 8's functional programming features. In Ruzica Piskac and Philipp Rümmer, editors, *Verified Software. Theories, Tools, and Experiments*, pages 267–278. Springer International Publishing, 2018. i

[15] 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. 5

[16] Michael Ernst and students. The Checker Framework.
https://checkerframework.org/manual/. 5, 22

[17] 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, iv

[18] Frama-C. https://frama-c.com. 5

[19] Patrice Godefroid. The soundness of bugs is what matters (position statement), 2005.
https://alastairreid.github.io/RelatedWork/papers/godefroid:bugs:2005/. 103

[20] Michael Hicks. What is soundness (in static analysis)?
http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/. 103

[21] Documentation for javac.
https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html#options. 28

[22] The KeY project. https://www.key-project.org. iii, 1, 5, 20

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

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

[25] 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

[26] K. Rustan M. Leino et al. Dafny github site. https://github.com/dafny-lang/dafny. Accessed September 2021. 1, 5

[27] K.R.M. Leino and K. Leino. *Program Proofs*. MIT Press, 2023. 1

[28] 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. 104

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

[30] 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/. 103

Index