Does your software do what it should?

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

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

DRAFT February 21, 2022

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

Foreword

Gary write this?

Preface

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

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

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

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

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

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

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

Contents

Chapter 1

Introduction to JML and OpenJML

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

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

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

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

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

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

Additional resources are listed in §1.3.

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

1.1 Why specify? Why check?

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

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

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

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

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

1.2 Background on OpenJML

OpenJML is a tool for checking that the source code of a Java program is consistent with specifications for that code written in the Java Modeling Language (JML). The tool parses and type-checks the specifications and performs static or run-time checking of the implementation code and the specifications. Because OpenJML is built on the Java OpenJDK 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 imple-

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

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

This list shows the functionality present or anticipated in OpenJML:

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

Check the above list against talks and publications

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

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

1.3 Other resources

There are several other useful resources related to JML and OpenJML:

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

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

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

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

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

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

1.4 Sources of Technology

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

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 stored in and available from a GitHub project: https://github.com/OpenJML.

1.6 Organization 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.

Needs rethinking

Chapter 2

Installation

2.1 Installing OpenJML

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

- Find the latest release of the highest number series, currently 16+, 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 textitOJ`/openjml`l, 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 textitOJ`/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 *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.
- The executable mac-setup, which turns off MacOS warnings about unknown executables, if necessary
- TODO: java and javac???
- 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.
- TODO: Reference manuals

2.3 Local customization

OpenJML can be customized to your local environment as described in §4.2. Local properties are specified in a openjml.properties file, stored in the same directory as openjml.jar 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. §??). 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

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

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

Chapter 3

The OpenJML Command-line Tool

3.1 The command-line

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 the path to the executable followed by space-separated arguments. Arguments that contain spaces should be enclosed in double-quotes. The shell interpreter and the OS being run will dictate other properties of the command-line, such as how and when variables are substituted, filename expansion is performed, and how file-system paths are written.
- The arguments themselves are either (relative or absolute) paths to files or options. An option may by followed by a value (if ti requires a value), which is then the next argument in the command-line. Relative paths are relative with respect to the current working directory (as given by `pwd`).
- Options begin with an initial hyphen character. Though it is now more common to have long option names begin with two hyphens and abbreviated names begin with one (as in `--help` and `-h`) and some javac options do have alternative double-hyphen version, OpenJML follows Open-

JDK and `javac`'s general practice by using just one hyphen.

- If an option appears more than once, then the values designated by later (to the right) appearances override earlier appearances; options that are not listed have default values.
- Default values can be set by properties and environment variables (cf. **??**), otherwise a built-in value is used.
- Options may have boolean or string values, though string values may be constrained to a specific format, such as a numeral.
- A boolean option (e.g. `-xyz`) is set to true by either `-xyz` or `-xyz=true`, set to false by either `-no-xyz` or `-xyz=false`; `-xyz=` resets the option to its default.
- A string option is required to have a value, which is specified either by `-xyz=`*`value`* (preferably for JML options) or `-xyz=`*`value`*. Java options may not use the = form. The form `-xyz=` resets the option to its default.

Each of the options is described later in this document.

Is the default the built-in or the property-specified value?

3.1.1 Files and Folders

Besides options, the Java compiler only allows files to be designated on the command-line. OpenJLM allows specifying folders using the `-dir` and `-dirs` options. The `-dir` *<directory>* option indicates that the *<directory>* value (an absolute or relative path) should be understood as a folder; all `.java` files recursively within the folder are included as if they were individually listed on the command-line. The `-dirs` option indicates that each one of the remaining command-line arguments is interpreted as either a source file (if it is a file with a `.java` suffix) or as a folder (if it is a folder) whose contents are processed as if listed on the command-line. Note that the `-dirs` option must be the last option.

As described later in §**??**, 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 can type-check `.jml` files as well as `.java` files if they are placed on the command-line. Doing so can be useful to check the syntax in a specific `.jml` file, but is usually not necessary: when a

Check and edit this as appropriate: can .jml files be checked standalone?

`.java` file is processed by OpenJML, the corresponding `.jml` file is automatically found (cf. ??) and checked.

3.1.2 Exit values

A command-line tool running in a shell interpreter is expected to emit an interger 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

Compiler warnings and verification failures will be reported as errors if the `-Werror` option is used. This may change an `EXIT_OK` or `EXIT_VERIFY` result to an `EXIT_ERROR` result.

The user may also use the `-verify-exit` option ?? to change an `EXIT_VERIFY` value to one of the other values in the list.

Chapter 4

OpenJML **Concepts**

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

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

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

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

Recall that the folders on the class and source paths represent the root of the package for that class. That is, a class `p.AA` (in package `p`) must have a class file

at `X/p/AA.class` with X on the classpath or a source file `Y/p/AA.java` with Y on the sourcepath. 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. Whenever a class, either source or compiled, is read into OpenJML, it will look for a corresponding specification file on the *specspath*, which is set by the **--specs-path** option. First, the full specspath is searched for the corresponding `.jml` file; if it is not found, then the specspath is searched again for a corresponding `.java` file. If still not found and the class was read from a source file on the command-line, then a `.jml` is looked for in the same folder as the `.java` file; if that is not found then the `.java` file from the command-line is used. If the class was not read from the command-line, then a default set of specifications is used.

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

- `classpath`: The OpenJML classpath is set using one of these alternatives, in priority order, with the system library always being added as well:
  - As the argument to the OpenJML command-line option **-classpath**
  - As the value of the Java property `org.jmlspecs.openjml.classpath`
  - As the value of the system environment variable `CLASSPATH`
  - As the default, which is the current working directory (plus the system library)
- `sourcepath`: The OpenJML sourcepath is set using one of these alternatives, in priority order:
  - As the argument of the OpenJML command-line option **-sourcepath**
  - As the value of the Java property `org.jmlspecs.openjml.sourcepath`
  - As the value of the OpenJML classpath (as determined above), without the system libraries (which are all `.class` files)
- `specspath`: The OpenJML specifications path is set using one of these alternatives, in priority order, with the locations of the system library specifications always appended:
  - As the argument of the OpenJML command-line option **-specspath**
  - As the value of the Java property `org.jmlspecs.openjml.specspath`
  - As 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 is used for all of these paths. A common prac-

tice is to simply use a single directory path, specified using the system CLASSPATH or on the command-line using **-classpath**, 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 included in the specifications path (unless the option **--no-internal-specs** is set). The **--no-internal-specs** allows a user to replace the full set of system library specifications with an alternate set. However it is generally more convenient to simply include the alternate set on the specspath, as specification files will then be found in the alternate set before the built-in set.

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

There are a number of common scenarios:

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

There are two complicated scenarios:

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

with the one later in the sourcepath appearing on the command-line

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

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

The OpenJML tool is controlled by a variety of options, just as many other tools are. The general rules about options are presented in §?? and the implemented options are described in detail later (cf. §??); 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. OpenJML uses properties for options that are typically characteristics of the local environment that vary among different users or different installations. But they can also be used to set initial values of options, so they do not need to be set on the command-line.

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

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

Check the reading of openjml.properties.

Then the value of any property whose name has the form `org.jmlspecs.openjml.`*option* is used to set the value of the *option* (leaving

off the initial 1 or 2 hyphens). Check that form; prop name is differnt than below And then, finally, the options given on the command-line override any previously given values.

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

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

The properties that are currently recognized are these:

- `openjml.defaultProver` - the value is the name of the prover (cf. §4.3) to use by default
- `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. §4.3)
- `org.openjml.`*option*, where *option* is the name of an OpenJML option (without any leading hyphens)

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

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

- `OPENJML_command=esc openjml tutorial/T_ensures2.java` — temporary change just for this line
- `export OPENJML\_command=esc; openjml tutorial/T\_ensures2.java`

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

— 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 `.properties` file.

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

Does the template file really have all of these?

4.3 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:

- `openjml.defaultProver` to give the name of a prover (e.g., `z3-4.3`)
- `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., `openjml.prover.z3-4.3=...`)

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

Currently, OpenJML works best with Z3 v4.3.1, which is shipped with OpenJML.

4.4 Conditional JML annotations

JML defines two mechanisms 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.
- In expressions, the term `key("`*key*`")`, is either a true or false Boolean literal, depending on whether the given *key* is defined or not

Each form relies on the *key* being defined or not. OpenJML defines keys using the `-keys` option, described in §??. Like other options, a property (`org.openjml.option.keys`) can be defined to avoid adding options to the command-line.

In OpenJML,

- the key `OPENJML` is enabled by default in the OpenJML tool
- the keys `ESC` and `RAC` are enabled when the respective OpenJML tools are being executed
- the key `DEBUG` is reserved but is disabled by default
- the key `KEY` is reserved for the use of the KeY ([17]) tool and is disabled by default in OpenJML
- all other keys are disabled by default in OpenJML

4.5 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`.

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

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 OpenJML is applied to a set of classes, by default it finds a version of the runtime classes and appends the location of the runtime classes to the classpath.

Review this about jmlruntime.jar

You can prevent OpenJML from automatically adding `jmlruntime.jar` to the classpath with the option **--no-internal-runtime**. If you use this option, then you will have to supply your own annotation classes and (if using Runtime Assertion Checking) your own runtime utility classes on the classpath. You may wish to do this, for example, if you have newer versions of the annotation classes that you are experimenting with. You could simply put them on the classpath, since they would be in front of the automatically added classes and used in favor of default versions; however, if you want to be sure that the default versions are not present, use the **--no-internal-runtime** option.

The symptom that no runtime classes are being found at all is error messages that complain that the `org.jmlspecs.annotation` package is not found.

CHeck that this is still true

4.6 Defaults for binary classes

TODO: Say more

4.7 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.

4.8 Nullness and non-nullness of references

TODO: Say more

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 §??. All of the options are listed alphabetically in Tables 5.1 and 5.2. The options are then described in following subsections in functionally similar groupings or in other chapters relevant to their functionality.

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

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

Options inherited from OpenJDK	
@*<filename>*	
-Akey	
--add-modules *<modulelist>*	[5.11]
-bootclasspath *<path>* --boot-class-path *<path>*	See Java documentation.
-cp *<path>* -classpath *<path>* --classpath *<path>*	location of input class files
-d *<directory>*	location of output class files
-deprecation	
--enable-preview	enables preview language features
-encoding *<encoding>*	
-endorsedirs *<dirs>*	
-extdirs *<dirs>*	
-g	generate debugging information
-h *<directory>*	location of generated header files
-? -help --help	output (Java and JML) help information
--help-extra	[5.7] help about extra options
-implicit	
-J*<flag>*	
--limit-modules *<modulelist>*	[5.11]
-m *<module>* --module *<module>*	[5.11]
--module-path *<path>*	[5.11]
--module-source-path *<path>*	[5.11]
--module-version *<version>*	[5.11]
-nowarn	show only errors, no warnings
-p *<path>*	like --module-path
-parameters	
-proc	
-processor *<classes>*	
--processor-module-path *<path>*	
-processorpath *<path>* --processor-path *<path>*	where to find annotation processors
-profile	
--release *<release>*	target release for compilation
-s *<directory>*	location of output source files
-source *<release>* --source *<release>*	the Java version of source files
-sourcepath *<path>* --source-path *<path>*	location of source files
--system *<jdk>*	

Options inherited from OpenJDK (cont.)	
-target *<release>* --target *<release>*	the Java version of the output class files
--upgrade-module-path *<path>*	[5.11]
-verbose	verbose output
-version --version	[5.7] output (OpenJML) version
-X	[5.7]Java non-standard extensions
-Werror	treat warnings as errors

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

Options specific to JML Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
--	no more options
[-]-benchmarks	outputs SMT files to use as benchmarks
[-]-check	[5.2] typecheck only (`-command=check`)
–check-accessible -checkAccessible	whether to check accessible clauses (default: true)
–check-feasibility *<list>* -checkFeasibility *<list>*	kinds of feasibility to check
-check-specs-path -checkSpecsPath	[5.4] warn about non-existent specs path entries
[-]-code-math *<mode>*	arithmetic mode for Java code (default: safe)
[-]-command *<action>*	[5.2] which action to do: `check esc rac compile`, default is `check`
[-]-compile	[5.2] typecheck JML but compile just the Java code (`-command=check`)
[-]-counterexample -ce	[5.5] show a counterexample for failed static checks
[-]-defaults *<list>*	enables various default behaviors TBD
[-]-determinism	EXPERIMENTAL: ???
--dir *<dir>*	[5.4] argument is a folder or file; enaqbles processing all .java files in a folder
--dirs	[5.4] remaining arguments are folders or files
[-]-esc	[5.2] do static checking (`-command=esc`)
--esc-bv -escBV	uses bit-vector arithmetic (default: false)
--esc-max-warnings *<n>* -escMaxWarnings	max number of verification errors to report in -esc
-escMaxWarningsPath	TBD? KEEP THIS?

Options specific to JML (cont.) Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
[-]-exec *<file>*	file path to prover executable
[-]-exclude *<patterns>*	paths to exclude from verification
[-]-extensions *<classes>*	comma-separated list of extensions classes and packages
[-]-inline-function-literal	EXPERIMENTAL ?
-internalRuntime	[5.4] add internal runtime library to classpath
-internalSpecs	[5.4] add internal specs library to specspath
-java	[5.2] use the native OpenJDK tool
-jml	[5.2] process JML constructs
-jmldebug	[5.7] very verbose output (includes -progress)
-jmltesting	changes some behavior for testing (default: false)
[-]-jmlverbose	[5.7] JML-specific verbose output
[-]-keys	[5.4] define keys for optional annotations
[-]-lang *<language>*	
[-]-logic *<name>*	name of SMT logic to use (default: ALL)
[-]-method *<patterns>*	methods to include in verification
--nonnull-by-default -nonnullByDefault	[5.4] values are not null by default
[-]-normal	[5.7] only outputs errors; no pother progress information
--nullable-by-default -nullableByDefault	[5.4] values may be null by default
[-]-osname *<name>*	os name to use in selecting prover (default: "" (auto))
[-]-progress	[5.7] outputs errors, warnings, progress and summary information
[-]-properties *<file>*	property file to read (value required)
[-]-prover *<name>*	prover to use (default: z3-4.3)
-purityCheck	[5.4] check for purity
[-]-quiet	[5.7] no informational output
[-]-rac	[5.2] compile runtime assertion checks (`-command=rac`)
--rac-check-assumptions -racCheckAssumptions	[5.6] enables (default on) checking assume statements as if they were asserts
--rac-compile-to-java-assert -racCompileToJavaAssert	[5.6] compile RAC checks using Java asserts
--rac-java-checks -racJavaChecks	[5.6] enables (default on) performing JML checking of violated Java features
-racMissingModelFieldRepSource	TBD
-racMissingModelFieldRepBinary	TBD
--rac-precondition-entry -racPreconditionEntry	TBD
--rac-show-source	[5.6] includes source location in RAC assertion failure messages

Options specific to JML (cont.) Options indicated with [-]-<name> may be spelled with either one or two hyphens, with two preferred	
-racShowSource	
[-]-require-white-space	whether white space is requires after an @ (default: false)
[-]-show	prints the details of source transformation (default: false)
--show-not-executable -showNotExecutable	warn if feature not executable, in -rac operations (default: TBD)
--show-not-implemented -showNotImplemented	warn if feature not implemented (default: TBD)
--show-options -showOptions	print the accumulated option settings (default:false)
--show-skipped -skipped	show methods whose proofs are skipped (default: TBD)
[-]-solver-seed	seed to pass on to the SMT solver (default: 0 - no seed)
[-]-spec-math *<mode>*	arithmetic mode for specifications (default: bigint)
--specs-path -specspath	[5.4] location of specs files
[-]-split	splits proof of method into sections
--stop-if-parse-errors -stopIfParseErrors	stop if there are any parse errors (don't do type checking or verification attempts)
-staticInitWarning	TBD
[-]-subexpressions	[5.5] show subexpression detail for failed static checks (default: false)
[-]-timeout *<seconds*	timeout for individual prover attempts (default: TBD)
[-]-trace	[5.5] show a trace for failed static checks (default: false)
[-]-triggers	enable SMT triggers (default: true)
-typeQuants	TBD
[-]-verboseness *<n>*	level of verbosness (0=quiet .. 4=jmldebug) (default: 1, -normal)
[-]-verify-exit *<n>*	exit code for verification errors (default: 6)
[-]-warn *<list>*	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 Options: Operational modes

These operational modes are mutually exclusive.

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

5.2 Options: JML tools

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

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

5.3 The -no-internalSpecs option.

As described in §4.5, this option turns off the automatic adding of the internal specifications library to the specspath. If you use this option, it is your responsibility to provide an alternate specifications library for the standard Java class library. If you do not you will likely see a large number of static checking warnings when you use Extended Static Checking to check the implementation code against the specifications.

The internal specifications are written for programs that conform to Java 1.8.

5.4 Options: OpenJML options applicable to all OpenJML tools

- **--dir** *<folder>* : abbreviation for listing on the command-line all of the .java files in the given folder and its subfolders (recursively); if the argument is a file, use it as is. The argument may also be a path expression containing wild-cards (* and ?); such arguments are expanded into a list of files by Java programmatic glob expansion.
- **--dirs** : treat all subsequent command-line arguments as if each were the argument to `--dir`
- **--specs-path** *<path>* : defines the specifications path, cf. section TBD
- **-keys** *<keys>* : the argument is a comma-separated list of JML keys (cf. the JML Reference Manual), used to conditionally enable selected annotations
- **-strictJML** : warns about any OpenJML extensions to standard JML FIX THIS
- **--show-not-implemented** : prints warnings about JML features that are ignored because they are not implemented; the default is disabled (silently ignoring such features).
- **--nullable-by-default** : sets the global default to be that all declarations are implicitly `@Nullable`, if they are not explicitly declared `@NonNull`
- **--nonnull-by-default** : sets the global default to be that all declarations are implicitly `@NonNull` (the default), if not explicitly declared `@Nullable`

- **-purityCheck** : turns on (default is off) purity checking for library methods (currently `-no-purityCheck` is recommended since the Java library specifications are not complete for `@Pure` declarations)
- **--check-specs-path** : if enabled, checks that each element (directory or jar files) of the *specspath* actually exists; if disabled, non-existent entries are silently ignored (default: disabled)

5.5 Options: Extended Static Checking

These options apply only when performing ESC:

- **--prover** *<prover>* : the name of the prover to use: one of `z3_4_3`, `z3_4_5`, `cvc4`, `yices2`
- **--exec** *<file>* : the path to the prover executable to use
- **--method** *<methodlist>* : a semicolon-separated list of method names to check (default is all methods in all listed classes). In order to disambiguate methods with the same name, the items in the list may be fully-qualified method names, may include signatures (containing fully-qualified type names), and may be regular expressions (cf. §8.2.3)
- **--exclude** *<methodlist>* : a semicolon-separated list of method names to exclude from checking (default is to exclude none). The format for the items in the list is the same as for **-method** (cf. §8.2.3)
- **--check-feasibility** *<where>* : checks feasibility of the program at various points — a comma-separated list of one of `none`, `all`, `exit`, `debug` [TBD, finish list, give default]
- **--esc-max-warnings** *<int>* : the maximum number of assertion violations to look for; the argument is either a positive integer or `All`; the default is `All`
- **--counterexample** : prints out a counterexample for failed proofs
- **--trace** : prints out a counterexample trace for each failed assert (includes `-counterexample`)
- **--subexpressions** : prints out a counterexample trace with model values for each subexpression (includes `-trace`)

5.6 Options: Runtime Assertion Checking

These options apply only when doing RAC:

- **--show-not-executable** : warns about the use of features that are not executable (and thus ignored); turn off with `--no-show-not-executable` What is the default
- **--rac-show-source** : enables including source code information in RAC error messages (default is enabled; disable with `--no-rac-show-source`)
- **--rac-check-assumptions** : enables checking `assume` statements as if they were asserts (default is enabled; disable with`--no-rac-check-assumptions`) Is this default correct?
- **--rac-java-checks** : enables performing JML checking of violated Java features (which will just proceed to throw an exception anyway) (default is enabled; disable with `--no-rac-java-checks`)
- **--rac-compile-to-java-assert** : compile RAC checks using Java asserts (which must then be enabled using `-ea`) (default is disabled; disable with `--no-rac-compile-to-java-assert`)

5.7 Options: JML Information and debugging

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

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

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

- **--silent** : only an exit code
- **--quiet** : no informational output, only errors and warnings

- --**normal** : (default) some informational output, in addition to errors and warnings
- --**progress** : prints out summary information as individual files are processed and proofs are attempted (includes `-normal`)
- --**verbose** : prints out verbose information about the Java processing in OpenJDK (does not include other OpenJML information)
- --**jmlverbose** : prints out verbose information about the JML processing (includes `-verbose` and `-progress`)
- --**jmldebug** : prints out (voluminous) debugging information (includes `-jmlverbose`)
- --**verboseness** *<int>* : sets the verboseness level to a value from -1 .. 4, corresponding to `-silent`, `-quiet`, `-normal`, `-progress`, `-jmlverbose`, `-jmldebug`

Other debugging options:

- -**show** : prints out rewritten versions of the Java program files for informational and debugging purposes

An option used primarily for testing:

- -**jmltesting**: adjusts the output so that test output is more stable

5.8 Java Options: Version of Java language or class files

- --**source** *<level>* : this option specifies the Java version of the source files, with values of `4`, ..., `17`, ... This controls whether some syntax features are permitted. The default is the most recent version of Java.
- --**target** *<level>* : this option specifies the Java version of the output class files (for compilation or RAC)

5.9 Java Options: Other Java compiler options applicable to OpenJML

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

These options control where output is written:

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

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

- **--class-path** or **-cp** or **-classpath**: the parameter gives the Java classpath to use to find referenced classes whose source files are not on the command-line (cf. section TBD)
- **--source-path** or **-sourcepath**: the parameter gives the sequence of directories in which to find source files of referenced classes that are not listed on the command-line (cf. section TBD)
- **-deprecation**: enables warnings about the use of deprecated features (applies to deprecated JML features as well)
- **--nowarn**: shuts off all compiler warnings, *including the verification failures produced by ESC* CHECK THIS
- **-Werror**: turns all warnings into errors, including compiler, JML typechecking and JML verification failures
- **-verbose**: turn on Java verbose output (does not control JML output)
- **-Xprefer:source** or **-Xprefer:newer**: when both a .java and a .class file are present, whether to choose the .java (source) file or the file that has the more recent modification time [TBD - check that this works]
- **--stop-if-parse-errors**: if enabled (disabled by default), processing stops after parsing if there are any parsing errors in any file; otherwise an attempt is made to further process successfully parsed files, though error cascades may occur (TBD - check this, check the default)

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)
- **-Akey**
- **-bootclasspath**
- **-encoding**
- **-endorsedirs**
- **-extdirs**
- **-g**
- **-implicit**
- **-J**
- **-X**... : Java's extended options

5.10 Java options related to annotation processing

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

5.11 Java options related to modules

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

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

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

Chapter 6

OpenJML extensions to JML

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

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

6.1 Specification statements

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

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

Check the above list

OpenJML adds these, described in succeeding subsections:

- `check` statement (§6.1.1)
- `show` statement (§6.1.2)
- `havoc` statement (§6.1.3)
- `halt` statement
- `split` statement
- `reachable` statement (§6.1.6)

6.1.1 `check` statement

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

Type checking requirements:

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

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

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

a tool should give two errors: one that the `check` statement is not provable and a second that there might be a null-dereference in the `c.value` expression. In contrast, if an `assert` were used instead, 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.

6.1.2 `show` statement

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

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

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

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

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.

6.1.3 `havoc` statement

Grammar:
<havoc-statement> `::=` **`havoc`** *<store-ref-expression>* `...` `:`

Grammar needed

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

Need example

6.1.4 `halt` statement

Grammar: *<halt-statement>* `::=` **`halt`** `:`

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 more and 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.

6.1.5 `split` statement

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

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

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

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

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

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

Only in the last case is the optional expression permitted.

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

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

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

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

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

6.1.6 `reachable` statement

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

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

The examples that follow are explained by the comments:

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

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

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

`reachable` statements in a method, the check is for each one of them individually; they are not required to all be reachable for the same initial state.

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

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

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

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

6.2 Modifiers

Write this; check against JMLRMv2

function, immutable, no_state, two-state, query secret strictly_pure inline infer

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

6.3 Other enhancements

post for old/pre declarations in specifications

\nonnullelements for collection classes

specification of lambda functions

reachable, unreachable

begin end markers

inline_loop

\exception \values

\reach

mutliple arguments for \invariant_for, \static_invariant_for

invariants method spec clause

use of for_example as feasibility

recommends-else

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

allow optional semicolons

functional form of \lbl

@Options modifier

Chapter 7

OpenJML tools — Parsing and Type-checking

7.1 Parsing

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

Parsing is affected by these options:

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

files that had Java annotatinos (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.

7.2 Type-checking JML specifications

The type-checking phase includes all of OpenJDK's name and type attribution for Java and OpenJML's extending that type-checking to thed `/jml` files and to ensuring that the `.jml` files match contents of the Java `.class` ort `.source` files.

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

```
openjml --check options files
```

or

```
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.

7.3 Command-line options for type-checking

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

- **-purityCheck**: enables (default on) checking for purity; disable with `-no-purityCheck`
- **-internalSpecs**: enables (default on) using the built-in library specifications; disable with `-no-internalSpecs`
- **-internalRuntime**: enables (default on) using the built-in runtime library; disable with `-no-internalRuntime`

Chapter 8

OpenJML tools — Static Checking (ESC) and Verification

Type-checking is performed automatically prior to ESC (Extended Static Checking). Thus ESC also depends on the information described in Chapters 3, 5 and 7, particularly including the command-line options relevant to type-checking and the discussion of class, source, and specification paths in §4.1.

8.1 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. §??). The ESC tool will result in one of four outcomes:

- It issues one or more static checking warnings.
- It finds no warnings through static checking and checks feasibility.
- It exhausts memory resources or alloted time.
- It encounters some internal bug.

These scenarios are discussed in the following subsections.

8.1.1 Finding static 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 `-maxEscWarnings` 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 nondeterministic.

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 **??**.

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

Give an example

8.1.2 Checking feasiblity

A run of OpenJML with `-esc` may find no warnings through static checking. In this case, the tool runs 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 will check that it is feasible to reach the beginning of the method body and to reach the exit point of the method. The beginning of the method body is not feasible if there is some contradiction within the preconditions and invariants.

The `--check-feasibility` option gives some control over the detail of feasibility checking. For example, the user may wish to have more fine-grained feasibility checking performed (at the cost of more execution time) in order to help debug the specifications or implementation.

Give an example

8.1.3 Timeouts and memory-outs

The underlying SMT solvers may report a time-out or memory exhaustion. One option is to increase the time out limit (with the `-timeout` option). An alternate recourse in this situation is to attempt to simplify the implementation or the specification. A time-out option to OpenJML is passed through to the underlying SMT solver for it to interpret according to its own implementation, so the user can do some experimentation. When running static checking on a whole group of methods, it is useful to use a somewhat short time-out value, so that particularly difficult methods do not unduly delay obtaining results for other methods.

Timeout used for each prover invocation

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).

8.1.4 Bugs

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

8.2 Options specific to static checking

8.2.1 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)

8.2.2 Choosing the solver used to check

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

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

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

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

- **-prover *prover***: the name of the prover to use: one of
  - `z3_4_3`: description of versions here and for each item
  - `z3_4_5`
  - `cvc4`
  - `yices2`
  - [TBD: expand list]
  - What to say for a compliant SMT solver
- **-exec *path***: the absolute path to the executable corresponding to the given prover
- **-boogie**: enables using boogie (-prover option ignored; -exec must specify the Z3 executable; *Not yet implemented*)

8.2.3 Choosing what to check

The default behavior is to check each method in each file and folder listed on the command-line (or selected in the GUI). The set of methods checked can be

Table 8.1: 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

constrained by these options. In particular the `-method` option is often used to constrain checking to a single method while that method or its specifications are being debugged.

- **-method** <*methodlist*> : a semicolon-separated list of method names to check (default is all methods in all listed classes)
- **-exclude** <*methodlist*> : a semicolon-separated list of method names to exclude from checking (default: no methods are excluded)

The `-method` and `-exclude` options interact as shown in Table 8.1; in summary, `-exclude` overrides `-method`.

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

Matching rules. The argument of the `-method` and `-exclude` options is a

semicolon-separated set of strings. A method *matches* if any one of the individual strings matches the name of the method. A match occurs if anyone of the following is true:

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

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

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

8.2.4 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.

- **-checkFeasibility *where***: checks feasibility of the program at various points: one of `none`, `all`, `exit` finish list, give default
- **-escMaxWarnings *int***: the maximum number of assertion violations to look for; the argument is either a positive integer or `All` (or equivalently `all`, default is `All`)
- **-trace**: prints out a counterexample trace for each failed assert
- **-subexpressions**: prints out a counterexample trace with model values for each subexpression
- **-counterexample** or **-ce**: prints out counterexample information

Provide more information and examples

8.2.5 Controlling output

ESC can take a while to run if operating on a large set of software. It is useful then to have good progress reporting and to control the output produced. The basic controls are the level of verbosity, in particular the `-progress` setting and the options described in the previous subsection (§8.2.4).

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

- **-progress** : so that the starting and completing each method is reported; these delineations also serve to associate ewarning and error reports with the method that produced them
- **-escMaxWarnings=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.
- **-checkFeasibility=exit** : in general the default value should be used to minimize computation time, but for an overarching run, just check feasibility of the exit point of the method to be sure the absence of warnings is not due to some contradictory requirements or axioms.
- 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.

Chapter 9

Runtime Assertion Checking

Runtime Assertion Checking is not yet operational in the Java 17ff version of OpenJML.

9.1 Compiling classes with assertions

Runtime-assertion checking (RAC) is accomplished by

- compiling your program with the regular Java compiler
- compiling some (or all) of your classes with RAC enabled
- running your program and observing whether any assertions are violated

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
- the OpenJML runtime library (`jmlruntime.jar`) must be included in the classpath

Thus `java -jar $OPENJML/openjml.jar -rac -cp ZZZZ:$OPENJML/jmlruntime.jar ...` instead of `javac -cp ZZZZ ...`, with a ; instead of a : on Windows systems, and with appropriate substitution of $OPENJML with the path to the OpenJML installation directory.

There are a few points to note:

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

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

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

- More? Label with the label that is used.

9.2 Options specific to runtime checking

9.2.1 --show-not-executable

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

9.2.2 --show-not-implemented

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

9.2.3 --rac-show-source

--rac-show-source: (default: enabled) includes source location in RAC warning messages. If this option is enabled then RAC assertion violation messages will include text from the source file indicating the location of the violation, in addition to the report of line number. The option can provide more helpful error information, but it also can considerably increase the size of the compiled classes. Thus, if the line numbers are adequate and the source text is not particularly needed, the user may wish to disable this option.

As an example, the input file

```
1 public class A {
2
3   public static void main(String... args) {
4     //@ assert args.length == 1;
5   }
6 }
```

when compiled with the command

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

and run with

```
java -cp ".;jmlruntime.jar"A
```

produces the output

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

If compiled with

```
java -jar openjml.jar --rac --no-rac-show-source A.java
```

the output is

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

9.2.4 --rac-check-assumptions

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

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

So this input file

```
public class A {

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

```

```
8   //@ requires i == 1;
9   //@ ensures \result == 20;
10  public static int m(int i) {
11    return 10;
12  }
13
14  //@ requires i == 0;
15  //@ ensures \result == 20;
16  public static int mm(int i) {
17    return 10;
18  }
19 }
```

when compiled with the command

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

and run with

```
java -cp ".;jmlruntime.jar"A
```

produces the output

```
1  A.java:4: JML precondition is false
2      m(args.length);
3       ^
4  A.java:8: Associated declaration: A.java:4:
5    //@ requires i == 1;
6        ^
7  A.java:8: JML precondition is false
8    //@ requires i == 1;
9        ^
10 A.java:16: JML postcondition is false
11   public static int mm(int i) {
12                      ^
13 A.java:15: Associated declaration: A.java:16:
14   //@ ensures \result == 20;
15       ^
16 A.java:5: JML postcondition is false
17     mm(args.length);
18       ^
19 A.java:15: Associated declaration: A.java:5:
20   //@ ensures \result == 20;
21       ^
```

The example output shows the preconditions and postconditions each being

checked twice, once by the caller and once by the callee, because both assumptions and assertions are checked at runtime.

However, if the example is compiled with

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

the output is

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

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

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

9.2.5 –rac-java-checks

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

code size, this option is disabled by default. However, the option is useful during testing, because then all violations of JML assertions are reported through OpenJML, so a test harness can uniformly detect and report violations during unit testing.

The discussion in §9.3 below is also important to when and how JML violations are reported.

As an example, the input file

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

when compiled with the command

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

and run with

```
java -cp ".;jmlruntime.jar"A
```

produces the output

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

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

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

the output is

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

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

9.2.6 --rac-compile-to-java-assert

--rac-compile-to-java-assert: (default: disabled) compiles RAC checks using Java asserts (which must then be enabled using `-ea`) during execution, instead of using `org.jmlspecs.utils.JmlAssertionError`. When this option is enabled, all assertion violation reporting is through Java assertion errors; that is, Option (C) in §9.3 is used despite any system properties. Furthermore, no reports will be generated at all at runtime unless the Java option `-ea` is enabled.

Need example

9.2.7 --rac-precondition-entry

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

[TBD - should this turn on -racCheckAssumptions?]

Complete the above and Need an example

9.3 Controlling how runtime assertion violations are reported

There are three ways in which a RAC-compiled program can report assertion violations. These can be controlled by properties set at the time the RAC-enabled program is *run* (not when it is *compiled*). Note that if the option **--rac-compile-to-java-assert** is enabled (§9.2.6) then option (C) below is compiled in at compile time, and the various runtime alternatives described here are no longer available.

- A) as messages printed to `System.out`. In this case the program will continue executing after printing the assertion violation and may possibly encounter and report additional violations. This reporting mechanism is the default and applies if neither property `org.jmlspecs.openjml.racexceptions` nor

`org.jmlspecs.openjml.racjavaassert` is defined while the program is executing. In this reporting mode, an additional useful system property is `org.jmlspecs.openjml.racshowstack`. If this property is defined, then the stack trace to an assertion violation is reported along with the violation message. This makes the output more verbose, but may make it easier to debug why a particular violation is occurring.

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

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

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

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

As an example, the input file

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

when compiled with the command

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

and run with

```
java -cp ".;jmlruntime.jar" A
```

produces the output

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

If compiled the same way but run with

```
java -cp ".;jmlruntime.jar"-Dorg.jmlspecs.openjml.racshowstack A
```

the output is

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

If compiled the same way but run with

```
java -cp ".;jmlruntime.jar" -Dorg.jmlspecs.openjml.racexceptions A
```

the output is

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

And if compiled the same way but run with

```
java -cp ".;jmlruntime.jar"-ea -Dorg.jmlspecs.openjml.racjavaassert A
```

the output is (Bad line numbers)

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

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

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

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

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

9.3.1 RAC FAQs

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

9.3.1.1 Uncompiled fields and methods

When model or ghost fields or methods of class B are used by class A and class A is compiled with RAC, but class B is not, runtime errors will occur. This happens because the content of B.class is just what is produced by the Java compiler

and does not have any JML fields or methods. No error occurs at compile time because OpenJML can see the declarations of JML fields and methods in class B; since Java compilation units (e.g., A and B separately) can be compiled separately, the system does not know until runtime that B has not been compiled with JML.

Make an exmaple

9.3.1.2 Non-executable or unimplemented features

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

Make example

9.3.1.3 Try blocks too large

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

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

Some tips to avoid this problem are these:

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

Chapter 10

Other OpenJML tools

10.1 Generating Documentation

This section will be added later.

10.2 Generating Specification File Skeletons

This section will be added later.

10.3 Generating Test Cases

This section will be added later.

10.4 Inferring specifications

This section will be added later.

Chapter 11

Limitations of OpenJML's implementation of JML

11.1 Soundness and Completeness

Much is made of the soundness and completeness claims of program analysis tools. In fact programs verifiers and bug finding tools use the terms *soundness* and *completeness* in different ways. One way to think about this question is in terms of the guarantees that a tool claims to make.[1] A tool can be said to be *sound* if the guarantee it makes actually holds. Currently OpenJML does not completely implement JML. The differences are explained in the following subsections.

Bug-finders Users looking for bugs waste time analyzing bug reports that are not actual bugs; that is, they want Q_2 in Fig. 11.1 to be empty, ideally. They are not so concerned that all bugs are reported (thaat is, that Q_3 is empty); rather

[1] Gary leavens suggested this approach to me

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

Figure 11.1: Combinations of the behavior of a program P and tool T concerning a bug at program location L

they need to find and fix the most bugs in a fixed amount of time[24, 14, 15]. Consequently the soundness goal for a bug-finder is this: any reported bug is a true bug. (Q_2 is empty). A secondary goal is completeness: all bugs are found (Q_3 is empty).

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

Tools cannot achieve both soundness and completeness. In practice some 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*[22]: recognizing that tools cannot be completely sound and carefully describing in what ways they are not. Thereby, practioners are aware of the capabilities and limitations of a tool.

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

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

Note at the start that all tools suffer from this potential unsoundness: tools may

have bugs in them that lead to missing actual errors. And little of sophisticated program analysis tools are actually verified themselves.

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

11.2.1 Non-conservative defaults

More - particularly about binary files

11.2.2 Unchecked assumptions

JML allows the introduction of unchecked assumptions as `assume` statements and `axioms`

11.2.3 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`.

11.2.4 Non-sequential Java

More

11.2.5 Reflection

JML does not provide language features to reason about reflection... More

11.2.6 Class loading

More

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

Also discuss – static initialization,

11.3.1 Clauses and expressions

- `measured_by`
- `duration`
- `working_space`
- `\space`

11.3.2 model import statement

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

In these two examples,

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

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

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

In

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

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

However,

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

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

code and JML annotations.

TBD - more to be said about .jml files

11.3.3 purity checks and system library annotations

Review and rewrite this

JML requires that methods that are called within JML annotations must be pure methods (cf. section TBD). OpenJML does implement a check for this requirement. However, to be pure, a method must be annotated as such by either `/*@ pure @*/` or `@Pure`. A user should insert such annotations where appropriate in the user's own code. However, many system libraries still lack JML annotations, including indications of purity. Using an unannotated library call within JML annotation will provoke a warning from OpenJML. Until the system libraries are more thoroughly annotated, users may wish to use the `-no-purityCheck` option to turn off purity checking.

Chapter 12

Contributing to OpenJML

Up to date information for OpenJML developers is found on the OpenJML GitHub wiki, at https://github.com/OpenJML/OpenJML. The same information is discussed here, as a snapshot at the time of publication.

The source programming language for OpenJML is Java.

12.1 GitHub

Need updating to match the revisions for Java17

The GitHub project named OpenJML (github.org/OpenJML) holds a number of related repositories:

- The OpenJML source code and related repositories
- 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)
- A number of other repositories that are related to JML, some of which are relevant to OpenJML.

The OpenJML project contains these interrelated git repositories, important for OpenJML development:

- OpenJML: contains the core software for OpenJML, including the modified

OpenJDK and the tests and tutorial demos for OpenJML
- JMLAnnotations: the source for the `org.jmlspecs.annotation` package
- Specs: the source for the JML specifications for the Java system library classes
- OpenJMLDemo: demo material for OpenJML
- OpenJML-UpdateSite: the update site for the Eclipse plug-in
- Solvers: binary instances of SMT solvers
- SMT-Solvers: an Eclipse feature plug-in containing the Solvers project, so the solvers can be distributed through an update site
- openjml.github.io: the repository holding the material for the OpenJML website at www.openjml.org
- jdk8u-dev-langtools: the most recent snapshot of OpenJDK development merged into the OpenJDK folder in the OpenJML repository

In addition, Say more about these

- openjml-installer
- try-openjml
- jml-lang.org

12.2 Maintaining the development wiki

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

12.3 Issues

Bugs, new feature requests, user problems and the like are recorded in the GitHub Issues tool for the project. The current set of issues is somewhat polluted by issues imported from the old Sourceforge site, so many of the issues do not concern OpenJML. In an attempt to sort them, issues identified as relevant to OpenJML are marked with the OpenJML tag. However new issues may not be marked with any tag. Despite its limitations, this tool is the record of bugs and of some

of the feature requests.

OpenJML does not use the project management features of GitHub.

12.4 Creating a development environment

The following instructions are current as of this writing and apply to the Java17ff version of the OpenJML code. The OpenJML project wiki on GitHub will contain any updates to this information.

The OpenJDK code, which OpenJML extends, changed significantly with the introduction of modiules to Java and their use by OpenJDK. Consequently the organization of the OpenJML code also changed significantly from the oler Java 8 version to the current Java 17 version.

To create a local working copy, perform the following clone commands in a new, empty directory (which we will refer to as *$WC*):

```
1 git clone https://github.com/OpenJML/OpenJML.git
2 git clone https://github.com/OpenJML/JMLAnnotations.git
3 git clone https://github.com/OpenJML/Specs.git
4 git clone https://github.com/OpenJML/Solvers.git
```

This will create the following directory structure in *$WC*:

- JmlAnnotations — the source for the JML annotations library
- OpenJML/OpenJDKModule — the modified source of OpenJDK that is OpenJML
- OpenJML/OpenJMLTests — the unit and functionality tests for OpenJML
- OpenJML/vendor — the vendor branch holding a pristine version of the OpenJDK code
- Specs — the JML specifications of the Java system libraries
- Solvers — binary executables of SMT solvers

12.5 Running tests

To be written

12.6 Running a development version of the GUI

To be written

12.7 Building and testing releases

To be written

12.8 Packaging a release

To be written

12.9 Maintaining the project website

The source material for the project website is maintained in the `openjml.github.io` repository. Developers responsible for the website should clone this repository locally. Any material committed and pushed to the remote git repository (on the master branch) will appear directly on the public facing website, after a slight delay. The repository is configured to respond to the www.openjml.org domain name (and also http://openjml.org).

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

12.10 Updating to newer versions of OpenJDK

To be written

Appendix A

Static warning categories

The various warnings issued by ESC or RAC are grouped into categories to make them easier to understand. These categories are listed and explained in the tables in this appendix.

- Assertions or verification conditions generated by the semantics of Java and JML are reported by either ESC or RAC. These are listed in Table A.1 Fix table - should be C.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 `-racCheckAssumptions`. These are listed in Table A.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 A.1: Static warnings about assertions. These warnings are reported in RAC if the given condition is found to be false when executing the program; the warnings are reported in ESC if the prover cannot prove the condition is always true.

Warning class	Description
`Accessible`	an expression uses memory locations that violate an `accessible` clause
`ArithmeticCastRange`	the argument for an arithmetic cast operation is out of range for the target type
`ArithmeticOperationRange`	the result of an arithmetic operation is out of range for its result type
`Assert`	an explicit assert cannot be proved valid
`AssumeCheck`	TBD - assumption?
`Assignable`	an assignment or method call violates an `assignable` clause
`Axiom`	TBD - assumption
`Callable`	a method call violates a `callable` clause
`Constraint`	a `constraint` clause is not proved valid as part of a method postcondition
`ExceptionalPostcondition`	an exceptional postcondition (`signals` clause) is not proved valid
`ExceptionList`	an exception is thrown that is not in the `signals_only` exception list
`Initially`	an `initially` clause is not valid as part of a constructor postcondition
`Invariant`	
`InvariantReenterCaller`	
`InvariantEntrance`	
`InvariantExit`	
`InvariantExceptionExit`	
`InvariantExitCaller`	
`LoopCondition`	
`LoopDecreases`	the value in a loop `decreases` clause does not decrease in a loop iteration
`LoopDecreasesNonNegative`	the value in a loop `decreases` clause is negative at the beginning of a loop iteration
`LoopInvariant`	a loop invariant is not valid after the body of a loop

Static warnings about assertions (cont.)	
Warning class	Description
`LoopInvariantAfterLoop`	a loop invariant is not valid on exit from the loop
`LoopInvariantBeforeLoop`	a loop invariant is not valid before the first iteration of the loop
`NullCheck`	
`NullField`	as part of the postcondition of a method, a class field declared non_null cannot be proved to be not null
`PossiblyBadCast`	a reference expression cannot be proved to have the type requested in the cast
`PossiblyBadArrayAssignment`	assignment of a reference to an array where the reference type is not a subtype of the underlying array index type (a Java `ArrayStoreException`)
`PossiblyNullDeReference`	an expression being dereferenced is null
`PossiblyNullField`	a NonNull field has a null value when checked as part of invariants CHECK
`PossiblyNullValue`	the value for a switch, throw, or synchronized statement is null
`PossiblyNegativeSize`	an array creation expression has a negative size
`PossiblyNegativeIndex`	the index of an array index operation is negative
`PossiblyToolargeIndex`	the index of an array index operation is larger or equal to the array length
`PossiblyNullUnbox`	a null reference is being unboxed to a primitive
`PossiblyNullAssignment`	a null value is being assigned to a NonNull location
`PossiblyNullInitialization`	a NonNull field or variable is being initialized with a null value
`PossiblyDivideByZero`	the denominator of a division operation is 0
`PossiblyLargeShift`	the shift amount in a left shift operation is larger or equal to the number of bits in the left-hand argument (this is not illegal in Java, but usually surprises users)
`Postcondition`	a postcondition (`ensures` clause) is not valid
`Precondition`	the composite precondition of a method being called cannot be proved valid
`Reachable`	there is no execution path to a `Reachable` statement (ESC only)
`Readable-if`	a field is read when the readable-if condition is not valid
`StaticInit`	invariants or non-nullness of fields cannot be proved valid in static initialization
`UndefinedBadCast`	within a JML expression, a reference expression cannot be proved to have the type requested in the cast

Static warnings about assertions (cont.)	
Warning class	Description
`UndefinedDivideByZero`	the denominator of a division operation is 0 in a JML expression
`UndefinedNegativeIndex`	the index of an array index operation is negative in a JML expression
`UndefinedNegativeSize`	the size of an array is negative in a JML expression
`UndefinedNullDeReference`	an expression being dereferenced is null in a JML expression
`UndefinedNullUnbox`	a null reference is being unboxed to a primitive in a JML expression
`UndefinedNullValue`	in a JML expression, an expression in a switch, throw or syncronized expression is null
`UndefinedPrecondition`	the precondition of a (pure) method being called in a JML expression does not hold
`UndefinedToolargeIndex`	the index of an array index operation is larger or equal to the array length in a JML expression
`Unreachable`	there is an execution path to a `unreachable` statement
`Writable-if`	a field is written when the readable-if condition is not valid

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

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

Bibliography

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

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

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

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

[5] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. an overview of the leon verification system: Verification by translation to recursive functions. 1

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

[7] David R. Cok. The jSMTLIB User Guide, 2013. https://smtlib.github.io/jSMTLIB/. 48

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

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

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

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

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

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

[14] Patrice Godefroid. The soundness of bugs is what matters (position statement), 2005. https://alastairreid.github.io/RelatedWork/papers/godefroid:bugs:2005/. 66

[15] Michael Hicks. What is soundness (in static analysis)? http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/. 66

[16] Documentation for javac. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javac.html#op
23

[17] The KeY project. https://www.key-project.org. 3, 6, 20

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

[19] Gary T. Leavens, David R. Cok, and Amirfarhad Nilizadeh. further lessons from the jml project. ii

[20] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David R. Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML reference manual. Available from http://www.jmlspecs.org, September 2009. 1, 5

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

[22] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44?46, January 2015. 66

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

[24] Yichen Xie, Mayur Naik, Brian Hackett, and Alex Aiken. Soundness and its role in bug detection systems (position paper), 2005. https://alastairreid.github.io/RelatedWork/papers/xie:bugs:2005/. 66

Index