Does your software do what it should?

Tutorial and user guide to specification and verification with the Java Modeling Language and OpenJML

David R. Cok

DRAFT December 25, 2016

Contents

Foreword

Gary write this?

Preface

Chapter 1

Introduction to specification and automatic checking

1.1 Why specify? Why check?

TODO

1.2 Background of verification, JML, and OpenJML

TODO

1.3 Organization of this document

Needs rethinking

This document addresses three related topics: how to read, write, and use specifications; the Java Modeling Language (JML) in which specifications are written; and the OpenJML tool that provides editing and checking support for Java programs using JML.

These three topics are best learned in an interleaved fashion. The tutorial section (Part I) does just this. It introduces the simpler topics of specification, using Java programs with JML as the specification language, and using OpenJML as the tool to aid editing and checking, with motivating examples. A reader new to JML or to using specifications will find this tutorial to be the easiest introduction to the topics of this book.

However, it is also useful to have a compact description of each of the JML language and the OpenJML tool. These descriptions are found in Parts **??** and II) respectively. After some introduction, a reader may well want to take a break from the tutorial to read through and experiment with the details of JML and OpenJML. Once a reader has graduated from the tutorial and is specifying and verifying new examples, the description of JML serves as a summary of the JML language and the description of OpenJML is the user guide and reference manual for the tool. These two parts stand on their own.

Part **??** contains information for those interested in contributing to the document. Contributions in the form of bug reports and experience reports with substantial use cases or experience in teaching are always welcome; this information can be shared directly with the developers or through the jmlspecs mailing list. Part **??**, however, contains information primarily of interest to those developing and extending the OpenJML code itself.

The final part of the document, Part **??**, describes details of how OpenJML translates the combination of Java and JML. This is not meant to be read through and is only intended for the reader interested in the detailed semantics of JML and the implementation of OpenJML.

FIXME - what about a mailing list

Chapter 2

Other resources

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

- `http://www.jmlspecs.org` is a web site describing current on JML, including references to many publications, other tools, and links to various groups using JML.
- `http://www.jmlspecs.org/OldReleases/jmlrefman.pdf` is the official reference manual for JML, though it sometimes lags behind agreed-upon changes that are implemented in tools. (FIXME - make a better link)
- `http://www.openjml.org` contains a set of on-line resources for OpenJML
- `http://jmlspecs.sourceforge.net/OpenJMLUserGuide.pdf` is the most current version of this document
- `http://jmlspecs.sourceforge.net/OpenJMLUserGuide.html` is an HTML version, with frames, of this document; `http://jmlspecs.sourceforge.net/OpenJMLUserGuide-onepage.html` is the same material in one large HTML page.
- The source code for OpenJML, the original JML tools, and some other JML projects is contained in the jmlspecs sourceforge project at `http://sourceforge.net/projects/jmlspecs`.

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

- Key - FIXME - need url
- The previous generation of JML tools prior to OpenJML is available at `http://www.jmlspecs.org/download.shtml`.
- FIXME - need others

Part I

Tutorial introduction to specifying and checking Java programs

This Part describes how to use JML and OpenJML using step-by-step explanations of concrete examples. The examples demonstrate the core concepts and syntax of JML and how to use OpenJML to check and debug specifications.

Author: David R. Cok

Acknowledgments

he and she and it

Chapter 3

Quick start to OpenJML

The details of installing and running OpenJML are presented in Part II. However, an installation of the tool is needed to work through the tutorial. Some impatient readers may also wish to have a quick installation of the tool prior to diving into the full description. This section provides initial installation and use instructions.

What about the installation of SMT solvers? and the openjml.properties file? Can we make all of that seamless for a quick installation?

3.1 Installing and using OpenJML on the command-line

OpenJML is available as a command-line tool and as an Eclipse plug-in. Complete the following steps to install the command-line tool:

- First, be sure that you have Java 1.8 as your default Java installation. You can check this using the command `java -version`.
- Create or identify a directory (folder) in which to place the installation. Let $OPENJML represent the path to this installation directory.
- Download into $OPENJML either the zip file at `http://jmlspecs.sourceforge.net/openjml.zip` or a gzipped tar file from `http://jmlspecs.sourceforge.net/openjml.tar.gz` (the content is the same).
- Within that directory (i.e., `cd $OPENJML`, either unzip (`unzip openjml.zip`) or untar (`tar xvzf openjml.tar.gz`) the downloaded file in place.
- The result should be a small number of files added to $OPENJML, particularly including `openjml.jar`.

OpenJML is used as a typical command-line tool:

```
java -jar $OPENJML/openjml.jar <options> <files>
```

Example commands will be shown throughout this tutorial. A full description of the options is given in Chapter **??**.

3.2 Installing and using the OpenJML plug-in in Eclipse

TODO

- First, be sure that you Java have 1.8 as your default Java installation. You can check this using the command `java -version` at a command-line prompt.
- Check that you have Eclipse Neon.1 or later installed on your machine, or install it from `TBD`.
- MORE NEEDED

If the installation is successful, you will see a new top-level menu named ‘JML’ and some new tool-bar items (a yellow coffeecup, ‘ESC’, and ‘RAC’). There are also additions to other menus throughout Eclipse.

More on how to use

3.3 The tutorial examples

Say something about installing and the organization of the tutorial files

Chapter 4

Some details

Where should this material go?

4.1 The form of JML annotations

Need a brief statement of the comment syntax

4.2 Disambiguating 'annotation'

Formal specifications for code are often called annotations; in this document we often use the term 'JML annotations' to refer to specifications written in JML. There is also a specific syntactic construct in Java called 'annotations': the interfaces labeled with '@' symbols that can modify various syntactic elements of Java. Thus the simple term 'annotation' can be ambiguous. The ambiguity is heightened by the fact that JML annotations, such as `/*@ pure */`, can be expressed as Java annotations, `@Pure`. Comment on needing to include the package for `@Pure` etc.

In this document, we will generally disambiguate the term 'annotation' as either 'JML annotation' or 'Java annotation'; if used alone, 'annotation' will generally mean a JML annotation. We will also often use the term 'JML specification' in place of 'JML annotation'.

4.3 Syntactic conflicts with @

For historical reasons, specifications are often written as structured programming language comments, with the `@` symbol denoting a comment containing specifications.

Java comments begin with either `//` or `/*`; those comments that contain JML specifications begin with `//@` or `/*@`, just like javadoc comments begin with `//*` or `/**`. Similarly, `//` or `/*` are also used for comments in C and C++; the ANSI-C Specification Language also uses `//@` or `/*@` to indicate comments containing specifications within a C program.

Unfortunately, since the `@` symbol is also used for Java annotations, the following problem can arise. Some Java code is written something like (the particular Java annotation and its content are irrelevant) this:

```
@SuppressWarning("...")
class X
```

and then the user comments out the Java annotation without any whitespace:

```
//@SuppressWarning("...")
class X
```

Now JML tools will interpret the `//@` as the beginning of a JML annotation that will generally have parsing errors.

If the user includes whitespace, as in

```
// @SuppressWarning("...") class X
```

there is then no problem. The workaround for this conflict is to edit the original Java source to include the whitespace. In some situations, placing all JML annotations in a `.jml` file may solve the problem; however, some tools, including OpenJML, may still parse the `.java` file, including the erroneous apparently-JML annotations, even though those annotations are ignored when a `.jml` file is present.

4.4 .jml files and .java files

TBD .jml files hide annotations in .java files, except those in body of methods - but this is discussed elsewhere as well. Perhaps omit it here.

Chapter 5

Pre- and Postconditions

In this chapter we will work through the first tutorial example, demonstrating various kinds of method specifications.

5.1 Writing method specifications

The example in Listing 5.1 implements a countdown timer. The constructor initializes the timer with a given number of minutes and hours. Each call to `tick()` decrements the timer one minute; `done()` returns true when the remaining time becomes zero. The getter functions `minute()` and `hour()` return the current values of the minutes and hours remaining. Other time categories, such as seconds, are omitted to keep the example short and simple.

Specifications for this class should explain the behavior to a reader without needing reference to the implementation. The easiest methods to start with are the two getter functions. These simply return as the result the values of the implementation fields. They can be specified as shown in Listing 5.2. There are a number of things to note here:

- The `ensures` clause states what will be true if the method terminates normally, that is, without throwing an exception.
- Here the `ensures` clause states that the returned result of the method, denoted by `\result`, is equal to the value of the `minute` or `hour` field, respectively.
- In addition, the `//@ pure` modifier, or equivalently, the `@Pure` annotation, indicates that the method is *pure*, that is, that it does not alter any memory locations present in the state before the method call (the *pre-state*). A pure method may alter variables local to the method implementation as those are not part of the pre-state (cf. §**??**). Specifying pure is needed; without it the default applies, which is that any memory location at all (in the pre-state) may be modified. That

Listing 5.1: A countdown timer class

```
public class Timer {

  public int minute;
  public int hour;

  // create a Timer with the given time remaining
  public Timer(int hours, int minutes) {
    hour = hours;
    minute = minutes;
  }

  public int minute() {
    return minute;
  }

  public int hour() {
    return hour;
  }

  // Decrement timer by one minute
  public void tick() {
    minute--;
    if (minute < 0) { minute = 59; hour--; }
  }

  // returns true when timer is at 0
  public boolean done() {
      return (minute == 0 && hour == 0);
  }
}
```

Listing 5.2: Specifying getter methods

```
//@ ensures \result == minute;
//@ pure
public int minute() {
  return minute;
}

//@ ensures \result == hour;
@Pure public int hour() {
  return hour;
}
```

Listing 5.3: Specifying the constructor

```
//@ ensures minute == minutes && hour = hours;
//@ pure
public Timer(int minutes, int hours) {
  minute = minutes;
  hour = hours;
}
```

Listing 5.4: Specifying the tick() method

```
/*@   requires minute > 0;
  @   assignable minute;
  @   ensures minute == \old(minute) - 1;
  @ also
  @   requires minute == 0;
  @   assignable minute, hour;
  @   ensures minute == 59 && hour == \old(hour) - 1;
  @*/
public void tick() { .. }
```

is not an issue for establishing that this method is consistent with its specifications. However, if the method is called by a client, after the call, the client must assume that any field visible to the method may have been modified, making further knowledge about the client's behavior essentially impossible. @Pure requires an import

The `done()` method's specifications are similar to those of the getter methods. Again, we just need an `ensures` clause stating what the result of the method is and a `pure` modifier.

The constructor is also simple to specify. In this case all that is needed is to state that the fields are initialized to the values of the constructor arguments, as shown in Listing 5.3. The constructor is also declared `pure`; a pure constructor is allowed to change the non-static fields of its own class, since those fields are not part of the pre-state.

The constructor and the methods discussed so far have no *precondition*; they may be called no matter what the values of the object's fields. Preconditions are expressed with `requires` clauses. The default `requires` clause, applicable when none is written, as in these cases, is `requires true;` ; such a precondition is always true.

Finally, we specify the `tick()` method. In this case there are two situations. If the current value of `minute` is greater than 0, then the value of `minute` can simply be decremented. If the current value of `minute` is 0, then it must be set back to 59 and the `hour` value decremented. This two-part specification is represented by two *behaviors*, separated by an `also` keyword, as shown in Listing 5.4.

These behaviors contain a few additional features. The first is the `requires` clause. There is one for each behavior; it is the *precondition* for the behavior: when the `requires` clause is true then the rest of the clauses in that behavior must hold; when the precondition is not true, the rest of the clauses need not hold. In this case the two behaviors are controlled by the two expected preconditions: one when `minute` is positive, and one when it is zero.

Second is the use of `\old`. The `\old` keyword indicates that its argument is to be evaluated in the pre-state of the method. So `hour == \old(hour) -1` says that the value of `hour` when the method terminates (that is, in the *post-state*) is one less than the value in the pre-state.

Listing 5.5: The countdown timer class with initial specifications

```
public class Timer {

  public int minute;
  public int hour;

  //@ ensures hour == hours && minute == minutes;
  /*@ pure */ public Timer(int hours, int minutes) {
    hour = hours;
    minute = minutes;
  }

  //@ ensures \result == minute;
  /*@ pure */ public int minute() {
    return minute;
  }

  //@ ensures \result == hour;
  public int hour() {
    return hour;
  }

  // Decrement timer by one minute
  /*@   requires minute > 0;
    @   assignable minute;
    @   ensures minute == \old(minute) - 1;
    @ also
    @   requires minute == 0;
    @   assignable minute, hour;
    @   ensures minute == 59 && hour == \old(hour) - 1;
    @*/
  // Decrement timer by one minute
  public void tick() {
    minute--;
    if (minute < 0) { minute = 59; hour--; }
  }

  // returns true when timer is at 0
  //@ ensures \result == (minute == 0 && hour == 0);
  /*@ pure */ public boolean done() {
      return (minute == 0 && hour == 0);
  }
}
```

The last new feature is the `assignable` clause. This clause states which pre-state memory locations may be assigned in the course of executing the method; anything not listed is guaranteed to be unchanged. The locations listed are different for the two behaviors: in one case only `minute` is assigned; in the other, both `minute` and `hour` are altered.

Combining all of these specifications in one location give Listing 5.5.

5.2 Checking the specifications

With specifications written, we now need to check them. Two kinds of checks are needed. First we check that the specifications and the class methods are consistent;

then we also check that the specifications are useful for a client using the class. The specification writer can err in being too precise or in being insufficiently precise.

- **insufficiently precise**: Say a method's postcondition is simply `ensures true;`. This would be easily proved. Any implementation that terminated without exception would satisfy it. However, a client calling the method would know very little about the behavior of the method; little could be proved about the client's behavior.
- **too precise**: It seems counter-intuitive to think that a specification can be too precise. The problem here is in the limitations of tools. If a specification is very detailed, it will be more difficult to prove prove and use. It will also risk specifying the implementation rather than the intended behavior. If solvers can validate the specification and its uses, they may take more time on every check, slowing down the overall verification process. The goal is to specify just enough detail to adequately represent and verify the system as a whole.

5.2.1 Checking with the command-line tool

The specification above can be checked with the command

```
java -jar openjml.jar -esc -progress Timer.java
```

Check and fix the path to the demo files

The result is something like that shown in Figure 5.1. The output shows each of the four methods with a 'Completed proof' having no warnings. In addition it shows the result of selected feasibility checks. These checks are described in more detail later (§**??**).

Import the output from a file generated by running the example

Can't seem to get the environment working for boxed verbatim text

If you use the Eclipse plugin, then after loading the demo material, selecting the *demo1b/Timer.java* file, and invoking the tool bar item named `ESC`, the OpenJML console shows the material in Figure 5.2. Again, the output shows successful static checks for each method.

5.2.2 Checking a client

Now let's try to use this class. A simple client of `Timer` is shown in Listing 5.6.

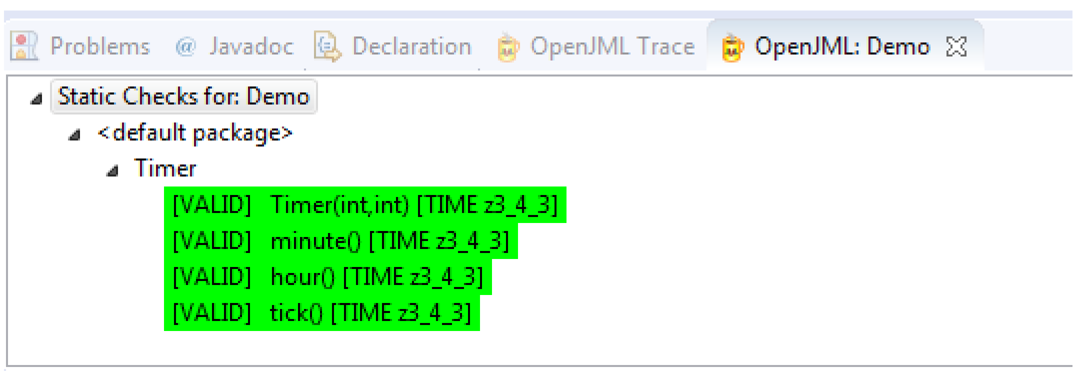
We run ESC on this class with the command

```
java -jar openjml.jar -cp .  -esc -progress Client.java
```

Figure 5.1: Output of commandline tool when checking Listing 5.5

```
Proving methods in Timer
Starting proof of Timer.Timer(int,int) with prover z3_4_4
Timer.java:7:  Feasibility check #1 - end of preconditions : OK
Timer.java:7:  Feasibility check #2 - at program exit : OK
Completed proof of Timer.Timer(int,int) with prover z3_4_4 - no warnings
Starting proof of Timer.minute() with prover z3_4_4
Timer.java:13:  Feasibility check #1 - end of preconditions : OK
Timer.java:13:  Feasibility check #2 - at program exit : OK
Completed proof of Timer.minute() with prover z3_4_4 - no warnings
Starting proof of Timer.hour() with prover z3_4_4
Timer.java:18:  Feasibility check #1 - end of preconditions : OK
Timer.java:18:  Feasibility check #2 - at program exit : OK
Completed proof of Timer.hour() with prover z3_4_4 - no warnings
Starting proof of Timer.tick() with prover z3_4_4
Timer.java:31:  Feasibility check #1 - end of preconditions : OK
Timer.java:31:  Feasibility check #2 - at program exit : OK
Completed proof of Timer.tick() with prover z3_4_4 - no warnings
Completed proving methods in Timer
```

Figure 5.2: GUI output when checking demo1b/Timer



Listing 5.6: A test client for Timer

```
public class Client {

  static Timer test(int h, int m) {
    Timer t = new Timer(h,m);
    t.tick();
    return t;
  }
}
```

This command contains `-cp .`; that option sets the classpath for finding `Timer.java` when referenced from Client.java. We could omit this option and instead list both `Timer.java` and `Client.java` on the command line, but then ESC would check both files (which sometimes may be what we want). The command produces output like that below.

```
Proving methods in Client
Starting proof of Client.Client() with prover z3_4_4
Client.java:1:  Feasibility check #1 - end of preconditions : OK
Client.java:1:  Feasibility check #2 - at program exit : OK
Completed proof of Client.Client() with prover z3_4_4 - no warnings
Starting proof of Client.test(int,int) with prover z3_4_4
Client.java:5: warning: The prover cannot establish an assertion (Precondition: Client.java:8: ) in method test
                t.tick();
                      ^
.\Timer.java:27: warning: Associated declaration: Client.java:5:
    @   requires minute == 0;
        ^
Completed proof of Client.test(int,int) with prover z3_4_4 - with warnings
Completed proving methods in Client
```

The default, empty constructor for `Client` is proved without a problem. But the proof of `test()` issues a warning that the precondition cannot be proved. If you want to obtain some detailed information about why this might be the case, add the option `-subexpressions` to the command and fairly voluminous tracing output will be produced. Skip down to where the Method Body starts and you will see output like the following:

```
        //Method Body
Client.java:4:          @NonNull  Timer t = new Timer(m, h)
                        VALUE: m         === ( - 2147483201 )
                        VALUE: h         === ( - 1 )
                        VALUE: new Timer(m, h)   === REF!val!11
                        VALUE: t         === REF!val!11
```

This indicates that ESC determined that the `test()` method does not behave according to specification when the inputs are -2147483201 and -1 for `m` and `h`. On reviewing `Timer.java`, we see that its constructor accepts these negative values without complaint, since its precondition is the default precondition, `requires true;`. However, `tick()` has a precondition and requires that at least one of `minute > 0` and `minute == 0` be true. That is, `tick()` is not implemented to support negative values of `minute`. In fact, we can see that although the implementation of `tick()` would terminate, the number of `tick()` calls until `done()` returned true might not be obvious to the caller.

There are a few solutions to this problem. One solution would be to expand our understanding of the desired behavior of `tick()` to include these negative values; we would then need to add additional behaviors to `tick()` to represent this additional behavior. A second solution, which we will adopt for the purposes of this tutorial, is to say that we want to forbid such negative values. In addition, we want to constrain `minute` to be in the range 0 to 59. In this case we need to do the following:

- restrict the inputs allowed to the constructor by adding a precondition
- optionally, we can state that the output of `minute()` is always in the range 0..59 and `hour()` is always non-negative
- `tick()` can presume that `minute` and shour are always in the expected range and must ensure that they are still in the expected range on output

These additional specifications are shown in Listing More to write

Additional sections: invariants, information hiding and datagroups; loop invariants; use of ghost fields to count ticks; use of assert statements; runtime checking, advanced features???

Should we have a section on common idioms and recommended style

Part II

The OpenJML tool for checking JML specifications

This Part is a user guide for using the OpenJML command-line and GUI tools to edit, review, and check Java code and JML specifications.

Author: David R. Cok

Acknowledgments

OpenJML was written primarily by David Cok. Other contributors are Gunjan Aggarwal, Daniel Houck, John Singleton, Dan Zimmerman. Many others contributed questions, bug reports and ideas, some through undergraduate, master's and Ph.D. projects built on or using OpenJML, others through using OpenJML for classroom teaching. OpenJML has also benefited from the many discussions on the definition of JML and of specification languages, such as with Gary Leavens, as the instigator of JML, and members of the KeY group.

Chapter 6

Introduction to OpenJML

OpenJML is a tool for processing Java Modeling Language (JML) specifications of Java programs. The tool parses and type-checks the specifications and performs static or run-time checking of the code and the 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 match informal or natural langauge specifications may also be necessary.

This list shows the functionality present or anticipated in OpenJML:

- parse and typecheck all of Java: Java parsed through Java 8, as implemented in OpenJDK
- all of Java with JML: Java modeled by JML as described in this book
- parse all of JML: all JML, as defined in this book, is parsed
- typecheck all of JML: most of JML is checked, as described in this book
- static checking that Java code is consistent with the JML specifications: implemented
- runtime checking of JML specifications: implemented
- interacting with OpenJML programmatically from a host program: implemented
- JML specifications included in javadoc documentation: planned
- JML specification inference: planned and in progress
- automatic test generation, based on JML specifications: planned

jmlxtodoCheck the above list against talks and publications

OpenJML can be used

- as a command-line tool to do type-checking and any of the functions listed above as implemented,
- as an Eclipse plug-in to perform those tasks, and
- programmatically from a user's Java program

OpenJML is constructed by extending OpenJDK, the open source Java compiler, to parse and include JML constructs in the abstract syntax trees created by the Java compiler to represent the Java program. OpenJDK, and thus OpenJML, is licensed under the GPLv.2, and consequently are freely available in source and binary form (with the restrictions on redistribution imposed by the GPL).

This Part describes how to use OpenJML. The details of how to write and understand JML specifications for Java programs are discussed in the Tutorial (Part I) and, in complete detail, in the JML Reference Manual (Part **??**).

- §**??**: How to install and run the command-line version of OpenJML
- §**??**: How to install and use the Eclipse plugin for OpenJML
- §**??**: OpenJML's options (command-line options and Eclipse preferences)
- §**??**: The runtime library
- §**??**: The specifications library
- §**??**: Organization of the GitHub source repository

6.1 Sources of Technology

The design and implementation of OpenJML uses and extends many ideas present in prior tools, such as ESC/Java[**?**] and ESC/Java2[**?**], and from discussions with builders of tools such as Spec#[**?**], Boogie[**?**], Dafny[**?**], Frama-C[**?**], KeY[**?**], and the Checker framework[**?**]. Some of the relevant published papers describing design aspects of tools like all of these are listed here:

- ESC/Java doc...
- paper on structure of verification conditions
- paper on efficient VCs
- paper on counterexamples
- P. Chalin, JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics, JOT, 3(6):57-79, 2004.
- Chalin on null types
- ...

Collect and add to these references

6.2 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.

The OpenJML plug-in is a pure Eclipse plug-in, and therefore is not required to be licensed under the EPL.

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`.

Chapter 7

Installing and running OpenJML

This chapter describes how to install, run and use the command-line version of OpenJML. The Eclipse plug-in version is described in §**??**.

7.1 System Requirements

7.1.1 Operating System

OpenJML is regularly tested in Linux (Ubuntu), Cygwin on Windows 7, Windows 7, and MacOS X (TBD) environments. As the tool is a pure Java application, we expect it to work well in other environments also. Feedback about success or failure in any environment is welcome.

7.1.2 Java

The OpenJML command-line tool requires a Java 1.8 JRE. Current versions of Java can be downloaded from

`http://www.oracle.com/technetwork/java/javase/downloads/index.html`

or

`http://openjdk.java.net/install`

7.1.3 SMT solvers

You will need an SMT solver to perform static checking of JML specifications. Some options are

- Z3 4.4
- Z3 4.3.2
- CVC4

Yices? alt-ergo? Simplify? Licenses? download locations? provide instances?

7.2 Command-line tool download

What about the installation of SMT solvers? and the openjml.properties file? Can we make all of that seamless for a quick installation?

The OpenJML command line tool can be downloaded from
`http://jmlspecs.sourceforge.net/openjml.tar.gz` .

7.3 Installation

The command-line tool is supplied as a .zip or a .tar.gz file, downloadable from `http://jmlspecs.sourceforge.net/`. Download the file to a directory of your choice (referred to as *$OPENJML* subsequently) and unzip or untar it in place. It contains the following files:

- openjml.jar - the main jar file for the application
- jmlruntime.jar - a library needed on the classpath when running OpenJML's runtime-assertion-checking
- jmlspecs.jar - a library containing specification files
- openjml-template.properties - a sample file, which should be copied and renamed `openjml.properties`, containing definitions of properties whose values depend on your local system
- LICENSE.rtf - a copy of the modified GPL license that applies to OpenJDK and OpenJML
- OpenJMLUserGuide.pdf - this document

You must run OpenJML in a Java 1.8 JRE; it is not compatible with Java 1.9 or Java 1.7 and earlier versions.

You should ensure that the `jmlruntime.jar` and `jmlspecs.jar` files remain in the same folder as the `openjml.jar` file.

In addition to OpenJML itself, you will also need a SMT solver if you intend to use the static checking capability of OpenJML (cf. §7.1.3). The location of an SMT solver can be specified on the command-line or, more easily, in the `openjml.properties` file. For example, if the Z3 4.4 solver is located in your system at absolute location <*path*>, then include in the `openjml.properties` file the line

```
openjml.prover.z3_4_4=<path>
```

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

7.4 Running OpenJML

7.4.1 The Java command line

To run OpenJML, be sure that the `java` command uses a 1.8 JVM and use the following command line. Here *$OPENJML* designates the folder in which the `openjml.jar` and other installation files reside.

```
java -jar $OPENJML/openjml.jar <options> <files>
```

Here <*options*> and <*files*> are options and absolute or relative (with respect to the current working directory) paths to files or directories. As is typical for command-line tools and for Java tools, options and files may be intermingled; also, to be consistent with the OpenJDK tool, options begin with a single hyphen character. The valid options are listed in Table 7.1 and are described in subsections below.

The following command is currently a viable, but less preferred, alternative (there is no guarantee that the package location of `Main` will remain the same).

```
java -cp $OPENJML/openjml.jar org.jmlspecs.openjml.Main <options>
<files>
```

7.4.2 Exit values

When OpenJML runs as a command-line tool, it emits one of several values on exit:

- 0 (`EXIT_OK`) : successful operation, no errors, there may be warnings (including static checking 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

The symbolic names listed above are programmatically defined in `org.jmlspecs.openjml.Main` and used when executing OpenJML programmatically (cf. §??).

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

7.4.3 Files

In the command templates above, <*files*> refers to a list of `.java` files. Each file must be specified with an absolute file system path or with a path relative to the current working directory (in particular, not with respect to the classpath or the sourcepath).

are .jml files allowed?

You can also specify directories on the command line using the `-dir` and `-dirs` options. The `-dir` <*directory*> option indicates that the <*directory*> value (an absolute or relative path to a folder) should be understood as a folder; all `.java` or specification 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 section 7.4.4, 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 `.java` file is processed by OpenJML, the corresponding `.jml` file is automatically found (cf. **??**) and checked.

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

7.4.4 Specification files

JML specifications for Java classes (either source or binary) are written in files with a `.jml` suffix or are written directly in the source `.java` file. When OpenJML needs specifications for a given class, it looks for a `.jml` file on the specspath. If one is not found, OpenJML then looks for a `.java` file on the specspath. Note that this rule requires that source files (that have specifications you want to use) must be listed on the specspath. Note also that there need not be a source file; a `.jml` file can be (and often is) used to provide specifications for class files.

Previous versions of JML had a more complicated scheme for constructing specifications for a class involving refinements, multiple specification files, and various prefixes. This complicated process is now deprecated and no longer supported.

[TBD: some systems might find the first .java or .jml file on the specspath and use it, even if there were a .jml file later.]

Options specific to JML	
–	no more options
-check	[7.6] typecheck only (`-command check`)
-checkSpecsPath	[7.6] warn about non-existent specs path entries
-command <*action*>	[7.6] which action to do: check esc rac compile
-compile	[7.6] TBD
-counterexample	[7.6] show a counterexample for failed static checks
-dir <*dir*>	[7.6] argument is a folder or file
-dirs	[7.6] remaining arguments are folders or files
-esc	[7.6] do static checking (`-command esc`)
-internalRuntime	[7.6] add internal runtime library to classpath
-internalSpecs	[7.6] add internal specs library to specspath
-java	[7.6] use the native OpenJDK tool
-jml	[7.6] process JML constructs
-jmldebug	[7.6] very verbose output (includes -progress)
-jmlverbose	[7.6] JML-specific verbose output
-keys	[7.6] define keys for optional annotations
-method	
-nonnullByDefault	[7.6] values are not null by default
-normal	[7.6]
-nullableByDefault	[7.6] values may be null by default
-progress	[7.6]
-purityCheck	[7.6] check for purity
-quiet	[7.6] no informational output
-rac	[7.6] compile runtime assertion checks (`-command rac`)
-racCheckAssumptions	[7.6] enables (default on) checking assume statements as if they were asserts
-racCompileToJavaAssert	[7.6] compile RAC checks using Java asserts
-racJavaChecks	[7.6] enables (default on) performing JML checking of violated Java features

JML options, continued	
-racShowSource	[7.6] includes source location in RAC warning messages
-showNotImplemented	warn if feature not implemented
-specspath	[7.6] location of specs files
-stopIfParseErrors	stop if there are any parse errors
-subexpressions	[7.6] show subexpression detail for failed static checks
-trace	[7.6] show a trace for failed static checks

Options inherited from OpenJDK	
-Akey	
-bootclasspath <*path*>	See Java documentation.
-classpath <*path*>	location of input class files
-cp <*path*>	location of input class files
-d <*directory*>	location of output class files
-encoding <*encoding*>	
-endorsedirs <*dirs*>	
-extdirs <*dirs*>	
-deprecation	
-g	
-help	output (Java and JML) help information
-implicit	
-J<*flag*>	
-nowarn	show only errors, no warnings
-proc	
-processor <*classes*>	
-processorpath <*path*>	where to find annotation processors
-s <*directory*>	location of output source files
-source <*release*>	the Java version of source files
-sourcepath <*path*>	location of source files
-target <*release*>	the Java version of the output class files
-X	Java non-standard extensions
-verbose	verbose output
-version	output (OpenJML) version
-Werror	treat warnings as errors

Table 7.1: OpenJML options. See the text for more detail on each option.

7.4.5 Annotations and the runtime library

JML uses Java annotations as introduced in Java 1.6. Those annotation classes are in the package `org.jmlspecs.annotation`. In order for files using these annotations to be processed by Java, the annotation classes must be on the classpath. 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, as well as containing a version of the library within `openjml.jar`. 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.

You can prevent OpenJML from automatically adding `jmlruntime.jar` to the classpath with the option `-noInternalRuntime`. If you use this option, then you will have to supply your own annotation classes and (if using Runtime Assertion Checking) 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 `-noInternalRuntime` 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.

7.4.6 Command-line options

OpenJML's command-line options operate in a similar fashion to OpenJDK's options. Many command-line options have a corresponding Preference in the OpenJML Eclipse plug-in. Also, each option has a corresponding Java property; property files can be used to set options without needing to specify them on the command-line, effectively creating local default values. Property files are described in the next subsection (§7.4.7). Information about options, including their default values, can be obtained using the `-help` option.

- Options are identified by a leading hyphen (-) character.
- Arguments to options are given either as the next command-line argument or with the = syntax, as in either -*option value* or -*option*=*value*. If the argument is optional then only the = syntax may be used.
- If an argument contains space characters, it must be enclosed in double-quote characters, as is the case for any other command-line tool.
- If an option is repeated, the last occurrence overrides earlier ones.
- Each option has a default value.
- Boolean-valued options do not require but may have an argument.

  - `-`*option* enables the option
  - `-no-`*option* disables the option
  - `-`*option*`=true` enables the option
  - `-`*option*`=false` disables the option
  - `-`*option*= resets the option to its default System default? or default after properties are read?
- Other options typically require either integer or string arguments; each has a default value.
  - `-`*option*=*value* sets the value of the option
  - `-`*option value* sets the value of the option
  - `-no-`*option* resets the option to its default

7.4.7 Java properties and the `openjml.properties` file

OpenJML uses Java properties to define values specified outside the command-line. Java properties are typical key-value pairs of two strings. OpenJML properties are typically characteristics of the local environment that vary among different users or different installations. They can also be used to set initial values of options, so they do not need to be set on the command-line. An example is the file system location of a particular solver.

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 will supplant earlier ones.

- System properties, including those defined with `-D` options on the command-line
- The first `openjml.properties` file on the system classpath, if any
- A `openjml.properties` in the user's home directory (the value of the Java property `user.home`), if any
- A `openjml.properties` in the current working directory (the value of the Java property `user.dir`)
- Then any property whose name has the form `org.jmlspecs.openjml.`*option* is used to set the given *option* to the property's value. Check that form
- Finally, the options given on the command-line override any previously given values.

Check the reading of openjml.properties. SHould we have an installation wide copy?

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

- Lines that are all white space or the first non-whitespace character is a `#` or are comment lines
- Non-comment lines have the form *key*=*value* or *key*`:`*value*
- Whitespace is allowed between the key and the = or `:` character

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

- 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. §**??**) 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
- `org.jmlspecs.openjml.`*option*, where *option* is the name of an OpenJML option (without the leading hyphen)

[TBD: Check the above]

Need to talk about SMT provers somewhere

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

7.5 Options: Finding files and classes: class, source, and specs paths

Duplicated in section 9.2

A common source of confusion is the various different paths used to find files, specifications and classes in OpenJML. OpenJML is a Java application and thus a *classpath* is used to find the classes that constitute the OpenJML application; but OpenJML is also a tool that processes Java files, so it uses a (different) classpath to find the files that it is processing. As is the case for other Java applications, a <*path*> contains a sequence of individual paths to folders or jar files, separated by the path separator character (a semicolon on Windows systems and a colon on Unix and MacOSX systems). You should distinguish the following:

- the classpath used to run the application: specified by one of
  - the `CLASSPATH` environment variable
  - the .jar file given with the `java -jar` form of the command is used
  - the value for the `-classpath` (equivalently, `-cp`) option when OpenJML is run with the `java -cp openjml.jar org.jmlspecs.openjml.Main` command

  This classpath is the path that Java users will be familiar with. The value is implicitly given in the `-jar` form of the command. The application classpath is explicitly given in the alternate form of the command, and it may be omitted; if it is omitted, the value of the system property `CLASSPATH` is used and it must contain the `openjml.jar` library.

- the classpath used by OpenJML. This classpath determines where OpenJML will find .class files for classes referenced by the `.java` files it is processing. The classpath is specified by

  `-classpath` <*path*>

  or

  `-cp` <*path*>

  after the executable is named on the commandline. That is,

  ```
  java -jar openmjml.jar -cp <openjml-classpath> ...
  ```

  If the OpenJML classpath is not specified, its value is the same as the application classpath.

- the OpenJML sourcepath - The sourcepath is used by OpenJML as the list of locations in which to find `.java` files that are referenced by the files being processed. For example, if a file on the command-line, say `T.java`, refers to another class, say `class U`, that is not listed on the command-line, then `U` must be found. OpenJML (just as is done by the Java compiler) will look for a source file for `U` in the sourcepath and a class file for `U` in the classpath. If both are found then TBD.

  The OpenJML sourcepath is specified by the `-sourcepath` <*path*> option. If it is not specified, the value for the sourcepath is taken to be the same as the OpenJML classpath.

  In fact, the sourcepath is rarely used. Users often will specify a classpath containing both `.class` and `.java` files; by not specifying a sourcepath, the same path is used for both `.java` and `.class` files. This is simpler to write, but does mean that the application must search through all source and binary directories for any particular source or binary file.

- the OpenJML specspath - The specspath tells OpenJML where to look for specification (`.jml`) files. It is specified with the `-specspath` <*path*> option. If it is not specified, the value for the specspath is the same as the value for the sourcepath. In addition, by default, the specspath has added to it an internal library of specifications. These are the existing (and incomplete) specifications of the Java standard library classes.

  The addition of the Java library specifications to the specspath can be disabled by using the `-noInternalSpecs` option. For example. if you have your own set of specification files that you want to use instead of the internal library, then you should use the `-noInternalSpecs` option and a `-specspath` option with a path that includes your own specification library.

  Note also that often source (`.java`) files contain specifications as well. Thus, if you are specifying a specspath yourself, you should be sure to include directories containing source files in the specspath; this rule also includes the `.java` files that appear on the command-line: they also should appear on the specspath.

TBD - describe what happens if the above guidelines are not followed. (Can we make this more user friendly).

7.6 OpenJML options

There are many options that control or modify the behavior of OpenJML. Some of these are inherited from the Java compiler on which OpenJML is based. Options for the command-line tool are expressed as standard command-line options. In the Eclipse GUI, the values of options are set on a typical Eclipse preference or properties page for OpenJML. Should we leave out these references to the GUI here and elsewhere?

Options: Operational modes

These operational modes are mutually exclusive.

- textbf-jml (default) : use the OpenJML implementation to process the listed files, including embedded JML comments and any corresponding `.jml` files
- textbf-no-jml: uses the OpenJML implementation to type-check and possibly compile the listed files, but ignores all JML annotations in those files
- textbf-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.

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.

- textbf-command <*tool*> : initiates the given function; the value of <*tool*> may be one of `check`, `esc`, `rac`, `doc`. The default is to use the OpenJML tool to do only typechecking of Java and JML in the source files.
- textbf-check : causes OpenJML to do only type-checking of the Java and JML in the input files (alias for `-command=check`)
- textbf-compile : TBD
- textbf-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`)

- textbf-rac : compiles the given Java files as OpenJDK would do, but with JML checks included for checking at runtime (alias for `-command=rac`)
- textbf-doc : executes javadoc but adds JML specifications into the javadoc output files (alias for `-command=doc`) *Not yet implemented.*

Fix the following

The textbf-noInternalSpecs option. As described above, 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 1.7. [TBD - change this to adhere to the `-source` option?] [TBD - what about the specs in jmlspecs for different source levels.]

Options: OpenJML options applicable to all OpenJML tools

- textbf-dir <*folder*> : abbreviation for listing on the command-line all of the .java files in the given folder, and its subfolders; if the argument is a file, use it as is
- textbf-dirs : treat all subsequent command-line arguments as if each were the argument to `-dir`
- textbf-specspath <*path*> : defines the specifications path, cf. section TBD
- textbf-keys <*keys*> : the argument is a comma-separated list of options JML keys (cf. section TBD)
- textbf-strictJML : warns about an OpenJML extensions to standard JML

Check capitalization of the following

- textbf-nullableByDefault : sets the global default to be that all declarations are implicitly `@Nullable`
- textbf-nonnullByDefault : sets the global default to be that all declarations are implicitly `@NonNull` (the default)
- textbf-purityCheck : turns on (default is on) purity checking (recommended since the Java library specifications are not complete for `@Pure` declarations)
- textbf-checkSpecsPath : TODO

Check the following

- -Werror
- -nowarn

- -stopIfParseError

Options: Extended Static Checking

These options apply only when performing ESC:

- textbf-prover <*prover*> : the name of the prover to use: one of z3_4_3, cvc4, yices2
- textbf-exec <*file*> : the path to the executable corresponding to the given prover
- textbf-boogie : enables using boogie (-prover option ignored; -exec must specify the Z3 executable for Boogie to use)
- textbf-method <*methodlist*> : a comma-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 and may include signatures (containing just fully-qualified type names)
- textbf-exclude <*methodlist*> : a comma-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**.
- textbf-checkFeasibility <*where*> : checks feasibility of the program at various points — a comma-separated list of one of `none`, `all`, `exit` [TBD, finish list, give default]
- textbf-escMaxWarnings <*int*> : the maximum number of assertion violations to look for; the argument is either a positive integer or `All`; the default is `All`
- textbf-counterexample : prints out a counterexample for failed proofs
- textbf-trace : prints out a counterexample trace for each failed assert (includes -counterexample)
- textbf-subexpressions : prints out a counterexample trace with model values for each subexpression (includes -trace)

Options: Runtime Assertion Checking

These options apply only when doing RAC:

- textbf-showNotExecutable : warns about the use of features that are not executable (and thus ignored); turn off with `-no-shownotExecutable`
- textbf-showRacSource : enables including source code information in RAC error messages (default is enabled; disable with `-no-showRacSource`)
- textbf-racCheckAssumptions : enables checking `assume` statements as if they were asserts (default is enabled; disable with`-no-racCheckAssumptions`)

- textbf-racJavaChecks : enables performing JML checking of violated Java features (which will just proceed to throw an exception anyway) (default is enabled; disable with `-no-racJavaChecks`)
- textbf-racCompileToJavaAssert : compile RAC checks using Java asserts (which must then be enabled using `-ea`) (default is disabled; disable with `-no-racCompileToJavaAssert`)

Options: JML Information and debugging

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

- textbf-help : prints out help information about the command-line options
- textbf-version : prints out the version of the OpenJML tool software

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

- textbf-quiet : no informational output, only errors and warnings
- textbf-normal : (default) some informational output, in addition to errors and warnings
- textbf-progress : prints out summary information as individual files are processed (includes -normal)
- textbf-verbose : prints out verbose information about the Java processing in OpenJDK (does not include other OpenJML information)
- textbf-jmlverbose : prints out verbose information about the JML processing (includes -verbose and -progress)
- textbf-jmldebug : prints out (voluminous) debugging information (includes -jmlverbose)
- textbf-verboseness <*int*> : sets the verboseness level to a value from 0 - 4, corresponding to -quiet, -normal, -progress, -jmlverbose, -jmldebug

Other debugging options:

- textbf-show : prints out rewritten versions of the Java program files for informational and debugging purposes
- textbf-showNotImplemented : prints warnings about JML features that are ignored because they are not implemented; the default is disabled.

An option used primarily for testing:

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

Java Options: Version of Java language or class files

- textbf-source <*level*> : this option specifies the Java version of the source files, with values of `1.4`, `1.5`, `1.6`, `1.7`... or `4`, `5`, `6`, `7`, ... This controls whether some syntax features (e.g. annotations, extended for-loops, autoboxing, enums) are permitted. The default is the most recent version of Java, in this case 1.8. Note that the classpath should include the Java library classes that correspond to the source version being used.
- textbf-target <*level*> : this option specifies the Java version of the output class files

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:

- textbf-d <*dir*> : specifies the directory in which output class files are placed; the directory must already exist
- textbf-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.

- textbf-cp or textbf-classpath: the parameter gives the classpath to use to find unnamed but referenced class files (cf. section TBD)
- textbf-sourcepath: the parameter gives the sequence of directories in which to find source files for unnamed but referenced classes (cf. section TBD)
- textbf-deprecation: enables warnings about the use of deprecated features (applies to deprecated JML features as well)
- textbf-nowarn: shuts off all compiler warnings, *including the static check warnings produced by ESC*
- textbf-Werror: turns all warnings into errors, including JML (and static check) warnings
- textbf`@`*filename*: the given *filename* contains a list of arguments
- textbf-source: specifies the Java version to use (default 1.7)
- textbf-verbose: turn on Java verbose output
- textbf-Xprefer:source or textbf-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]
- textbf-stopIfParseErrors: if enabled (disabled by default), processing stops after parsing if there are any parsing errors (TBD - check this, describe the default)

Other Java options, whose meaning and use is unchanged from javac: DUplicated text?

- textbf@<*filename*> : reads the contents of <*filename*> as a sequence of command-line arguments (options, arguments and files)
- textbf-Akey
- textbf-bootclasspath
- textbf-encoding
- textbf-endorsedirs
- textbf-extdirs
- textbf-g
- textbf-implicit
- textbf-J
- textbf-nowarn : only print errors, not warnings, *including not printing static check warnings*
- textbf-Werror : turns all warnings into errors
- textbf-X... : Java's extended options

These Java options are discussed elsewhere in this document:

- textbf-cp <*path*> or textbf-classpath <*path*> : section 7.5
- textbf-sourcepath <*path*> : section 7.5
- textbf-verbose : section 7.6
- textbf-source :
- textbf-target :

7.6.1 Java options related to annotation processing

- textbf-proc
- textbf-processor
- textbf-processorpath

Check that the option lists are comprehensive, and up to date with Java 1.8

Chapter 8

The Eclipse Plug-in

Since OpenJML operates on Java files, it is natural that it be integrated into the Eclipse IDE for Java. OpenJML provides a conventional Eclipse plug-in that encapsulates the OpenJML command-line tool and integrates it with the Eclipse Java development environment. The plug-in also provides GUI functionality for working with JML specifications.

The Update site for the Eclipse plug-in that encapsulates the OpenJML tool is

```
http://jmlspecs.sourceforge.net/openjml-updatesite
```

8.1 System Requirements

Your system must have the following:

- A Java 1.8 JRE as described in section 7.4. This must be the JRE in use in the environment in which Eclipse is invoked. If you start Eclipse by a command in a shell, it is straightforward to make sure that the correct Java JRE is defined in that shell. However, if you start Eclipse by, for example, double-clicking a desktop icon, then you must ensure that the Java 1.8 JRE is set by the system at startup.
- Eclipse 4.4 (Neon) or later. (Java 1.8 requires Neon or later.)
- One or more SMT solvers, if the static checking functionality will be used.

List suitable solvers

8.2 Installation

Installation of the plug-in follows the conventional Eclipse procedure.

- Invoke the "Install New Software" dialog under the Eclipse "Help" menubar item.
- "Add" a new location, giving the URL `http://jmlspecs.sourceforge.net/openjml-updatesite` and some name of your choice (e.g. OpenJML).
- Select the "OpenJML" category and push "Next"
- Proceed through the rest of the wizard dialogs to install OpenJML.
- Restart Eclipse when asked to obtain full functionality.

Note that the plugin is installed in the Eclipse installation. All workspaces that use the same installation of Eclipse will now have the OpenJML plugin available.

If the plug-in is successfully installed, the toolbar will contain a yellow coffee cup icon and a top-level menu will contain an item named **JML** (along with other menubar/toolbar items).

8.3 GUI Features

Note that the JML logo is a JML-decorated yellow coffee cup; this logo is associated with various GUI elements.

8.3.1 Commands

The OpenJML plug-in adds a number of commands. These are visible in the *Preferences»General»Keys* dialog. All the OpenJML commands explicitly added by the OpenJML plug-in are in the 'JML' category. Some commands are automatically added by Eclipse and are in other categories; for example, Eclipse automatically adds commands to open each individual Preference Page and each kind of View You can sort the table of commands by category and you can filter the table, in order to show just those commands related to JML. Also, this dialog allows binding a keyboard key-combination to a command, as is the case for all Eclipse commands.

The commands are listed in Table **??**, with forward references to more detailed discussion.

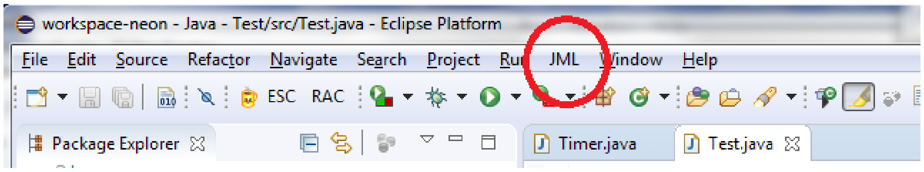
- Add to JML specs path (§**??**). Allows editing the specspath
- Clear All Results (§**??**). Deletes all results of static checking operations
- Clear Selected Results (§**??**). Deletes some of the results of static checking

- Delete JML Markers (§??). Deletes all JML markers and highlighting on selected resources
- Disable JML on the project (§??).
- Edit JML Source/Specs Paths (§??).
- Enable JML on the project (§??).
- Generate JML doc (§??).
- insert \result (§??).
- insert .... (§??).
- Open a Specifications Editor (§??).
- Open the ESC Results View (§??).
- RAC - ... (§??).
- Remove from JML specspath (§??).
- Rerun Static Check (§??).
- Show Counterexample (§??).
- Show Counterexample Value (§??).
- Show Detailed Proof Attempt Information (§??).
- Show ESC Result Information (§??).
- Show JML paths (§??).
- Show JML Specifications (§??).
- Static Check (ESC) (§??).
- Typecheck JML (§??). Performs syntax, parsing and typechecking on selected projects, folders, and files.
- Show In ... (§??).
- Show View (OpenJML Static Checks) (§??).
- Show View (OpenJML Trace) (§??).
- Preferences (OpenJML > OpenJML Solvers) (§??). Opens the OpenJML Solvers subpage. Preferences (OpenJML) (§??). Opens the OpenJML Preferences page.

8.3.2 Menubar additions

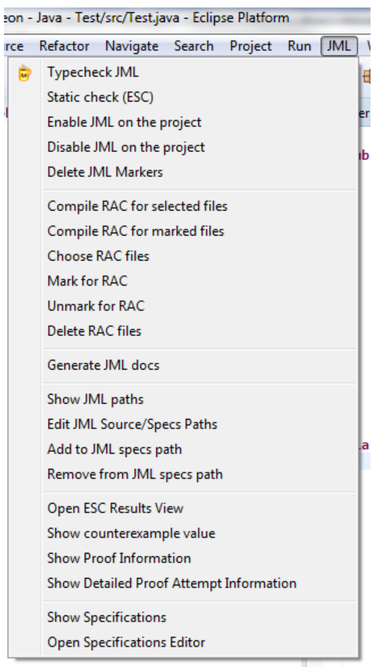
The main Eclipse menubar contains an additional menu titled 'JML', circled in red in Fig. 8.1. It contains various submenu items, as shown in Fig. 8.2; the action for a menu item is the similarly named command, described in §8.3.1. Individual menu items may

Figure 8.1: JML menu item on the Eclipse menubar



be disabled (grayed out) when they are not applicable. For instance, some items are enabled only when something is selected, some only when exactly one appropriate item is selected, some only when a method is selected, etc.

Figure 8.2: JML submenu items



The menu items (and the corresponding toolbar items described in §8.3.3) act on the contents of whichever files, folders or projects are *selected*. You can select multiple items by using the usual CTRL-click or SHIFT-click operations to extend the current selection with additional items. Files can be selected by making their editor windows active (i.e., by clicking on the desired editor window tab). Files, folders, projects, and individual functions can be selected in various Eclipse Views: Package Explorer, Project Explorer, Navigator, Outline. Different commands can operate on different entities. Commands that can operate on files will operate on all the files in selected folders or projects.

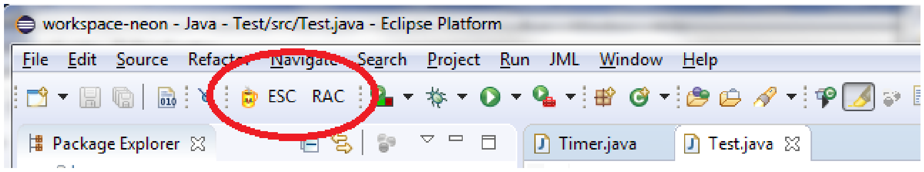
Menu items are also added in the following Context menus (context menus are available by right-clicking):

- Context menu on elements of the Eclipse Package Explorer View, Project Explorer View, and Navigator View
- Context menu on elements of the Outline View
- Context menu within an editor
- ... Problem View...
- ... OpenJML Proof view ...

8.3.3 Toolbar additions

The OpenJML plug-in adds three toolbar items, circled in red in Fig. 8.3; clicking the toolbar item executes a corresponding command. These are the same operations as the

Figure 8.3: JML additions to the Eclipse toolbar



corresponding items on the JML submenu and operate on the selected items as noted in §8.3.2.

- the JML coffee cup logo - executes the 'Typecheck JML' command, described further in §**??**
- ESC - executes the 'Static Check (ESC)' command, described further in §**??**
- RAC - executes the 'RAC - compile selected' command, described further in §**??**

8.3.4 OpenJML Problems, Markers and highlights

Eclipse uses *markers* to indicate the location of warnings and errors (generically, *problems*) in source files. They are typically shown along the left side of an editor pane and possibly as highlights or underscoring in the text itself. OpenJML defines a number of markers. They are typically a white 'J' superimposed on a colored disk. The various kinds of markers are shown in Table 8.1.

OpenJML problems are also listed in the Problems View along with other problems reported by the JDT or other plug-ins. OpenJML problems belong to a specific type, "JML Problem", so the Problem View can be filtered or sorted using that name as one criterion.

Eclipse permits the appearance of Problems in editor panes to be customized using general Preference settings. Navigate to the *General » Editors » Text Editors » Annotations* preferences page (in Eclipse for Windows — the location may be slightly different on other systems). Select an annotation type in the scrollable left-hand pane, such as one of the JML annotation types. Then the appearance settings for that annotation type can be altered on the right:

- whether the icon is shown in the vertical ruler (along the left edge of the editor pane)
- whether a navigation mark is shown in the overview ruler (along the right edge of the editor pane)
- whether the problem is indicated in the source text as well, as either highlighting or a squiggly underscore or not at all
- the color of the highlighting or squiggly underscore

Table 8.1: OpenJML markers and annotations

Annotation type	Purpose	Default highlighting
JML Error	parsing and type error	red with icon
JML Warning	parsing and type warning	yellow with icon
JML Note	informational note	blue with icon
JML Static Check Warning	static checking warning	orange with icon
JML Execution path	counterexample path	yellow
JML Execution path - True	true conditions on path	green
JML Execution path - False	false conditions on path	red
JML Execution path - Exception	exceptions on path	orange

The icon associated with an annotation type (or its color) cannot be changed using preferences.

The annotations in the lower part of Table 8.1 are used in showing counterexample paths (cf. §**??**). If supported by the underlying SMT solver, each static check warning is accompanied by a counterexample that shows how the assertion being warned about is not true. The counterexample consists of a path through the code and values for each variable and subexpression along the path. The statements along the path are highlighted in yellow; any boolean conditions, such as branch conditions, assertions and postconditions, are highlighted green for true, red for false; and any execution paths resulting from an exception are highlighted orange. All the highlighting colors can be customized as described above.

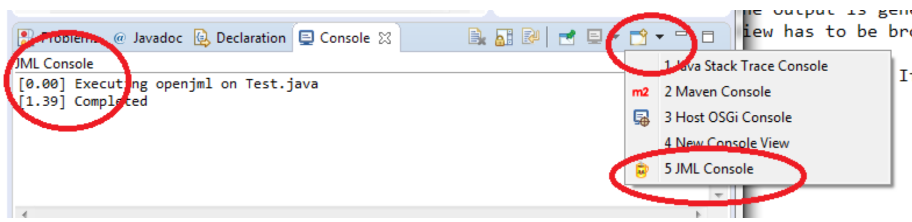
Need some screen shots of markers and counterexample highlighting

8.3.5 OpenJML console and error log

The OpenJML plug-in adds an additional type of console. General informational output is sent to the console; errors and warnings are placed in the console and shown in pop-up dialog boxes. Errors and warnings related to the source code itself are shown as Eclipse Markers (cf. §8.3.4) and recorded in the Console. Egregious errors are also logged in the Eclipse Error Log. In general, the Console will contain the information that the command-line tool would print out.

Eclipse has a Console View that manages the consoles for various plug-ins. In the OpenJML plugin, the Concole View's New Console menu has an additional item that creates a new OpenJML Console, as highlighted in Fig. 8.4. However, creating a OpenJML Console manually is rarely needed. The OpenJML Console is created automatically whenever output is generated by a JML operation. In addition, when output is generated the OpenJML Console will be made active (unless the user has locked the Console View), so the output is generally immediately obvious. When ESC is run, some other views (cf. §**??**) are also created, so in that case the Console View has to be brought to the front by hand.

Figure 8.4: The JML Console



Only one instance of a JML Console is ever created. Attempts to create another will just activate the existing one. If there is more than one Console View, they char the same instance of a JML Console.

8.3.6 Preferences

The plug-in adds dialogs for setting OpenJML options. These are workspace preferences, affecting all projects in the workspace. There are two Preference pages:

- A top-level page named 'OpenJML' found in the top-level list of Eclipse preference pages. This page allows setting options that would otherwise be set on the command-line. These is also one UI option, named 'UI verbose', that enables verbose output to the OpenJML console about actions within the UI code.
- A sub-page named 'Solvers'. (Click the turnstile next to 'OpenJML' in the list of Preference pages to see the subpage). These preferences enable registering SMT solvers and setting the default solver to use.

There are currently no project-level preferences within the OpenJML plug-in.

8.3.7 Editor embellishments

TBD - fill out

- counterexample hovers
- quick fix proposals
- context-sensitive completions
- insertions

8.3.8 OpenJML Views

TBD

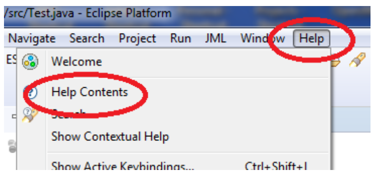


Figure 8.5: The Eclipse Help menu item

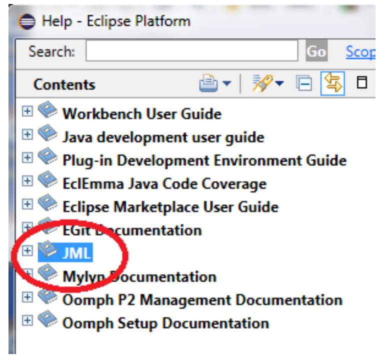


Figure 8.6: Eclipse Help table of contents

8.3.9 Help

There is a 'JML' entry in the table of contents under the Eclipse Help menu item. It provides an online user guide to JML and OpenJML. First click the *Help » Help Contents* menu item from the menubar, as shown in Fig. 8.5; then select 'JML' in the displayed Help table of contents as shown in Fig. 8.6 (the actual table of contents will vary depending on what plug-ins are present in your Eclipse installation).

Note - the current information available under Help is outdated and will be replaced by this manual.

8.3.10 Other GUI elements

- OpenJML decoration - A decoration is applied to names of projects in the Package Explorer View for which OpenJML has been enabled (cf. §**??**). The decoration is a miniature JML logo on the upper-right of the folder icon, covering the 'J' symbol that indicates an Eclipse Java project.
- The plug-in defines a JML Nature and a JML Builder. The Nature is associated with a project precisely when JML is enabled for the project. The Builder performs automatic type-checking. (cf. §**??**)
- `.jml` suffix. The plug-in adds a content type that associates the `.jml` filename suffix with the Java editor. This makes the Java editor the default editor for .jml files.

  Note - so we still get spurious errors on jml files?
- Internationalization. TBD...
- classpath intializer. TBD ...
- definition as an Eclipse project. TBD ...

- Open JML Perspective. TBD ...

Chapter 9

OpenJML tools

9.1 Parsing and Type-checking

The foundational function of OpenJML is to parse and check the well-formedness of JML annotations in the context of the associated Java program. Such checking includes conventional type-checking and checking that names are used consistently with their visibility and purity status.

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

```
java -jar $INSTALL/openjml.jar -check options files
```

or

```
java -jar $INSTALL/openjml.jar options files
```

since `-check` is the default action. The equivalent action in the Eclipse plug-in is the 'Typecheck JML' command, available through the toolbar or menu actions. Any `.jml` files are checked when the associated `.java` file is created. 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.

9.2 Classpaths, sourcepaths, and specification paths in OpenJML

Duplicated in chapter 7

A key concept to understand is how class files, source files, and specification files are found and used by the OpenJML tool. This process is described in the following subsection.

When a Java compiler parses source files, it considers three types of files:

- Source files listed on the command-line
- Other source files referenced by those listed on the command-line, but not on the command-line themselves
- Already-compiled class files

The OpenJML tool considers the same files, but also needs

- Specification files associated with classes in the program

The OpenJML tool behaves in a way similar to a typical Java compiler, making use of three directory paths - the classpath, the sourcepath, and the specspath. These paths are standard lists of directories or jar files, separated either by colons (Unix) or semicolons (Windows). Java packages are subdirectories of these directories.

- `classpath`: The OpenJML classpath is set using one of these alternatives, in priority order:
  - 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`
- `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)
- `specspath`: The OpenJML specifications path is set using one of these alternatives, in priority order:
  - 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 practice is to simply use a single directory path, specified 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-internalSpecs` is set).

CHeck the spelling of -no-internalSpecs

The paths are used as follows to find relevant files:

- Source files listed on the command-line are found directly in the file system. If the command-line element is an absolute path to a `.java` file, it is looked up in the file system as an absolute path; if the command-line element is a relative path, the file is found relative to the current working directory.

Java source	Java byte-code	JML specs	
command-line	none	specspath	Use JML as specs for Java source
command-line	none	none	Use Java source as its own specs
none	classpath	specspath	Use JML as specs for byte-code
none	classpath	none	Use default specs for byte-code

- Classes that are referenced by files on the command-line or transitively by other classes in the program, can be found in one of two ways:
  - The source file for the class is sought as a sub-file of an element of the `sourcepath`.
  - The class file for the class is sought as a sub-file of an element of the `classpath`.

  If there is both a sourcefile and a classfile present, then
  - if the option `-Xprefer:source` is present, the sourcefile is always recompiled
  - if the option `-Xprefer:newer` is present, the sourcefile is recompiled only if its modification timestamp is newer than that of the class file.

  The default is to use the newer of the source or class files.

The JML specification files associated with Java source or class files are found as follows:

- The specifications path as determined above is augmented with the built-in libraries specifications (unless the option `-no-internalSpecs` is operative).
- For each Java class (whether in source or byte-code) the corresponding `.jml` file is found by searching the specifications path using the fully-qualified (package+class name) of the class. The first match to the fully-qualified class name is used.
- If no specifications file is found, then the Java source file for that class is used as the specifications file. This would typically be the same file as is compiled as the `.java` file.
- If no specifications or source file is found, then the byte-code class is used with default JML specifications.

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. what about recompiled cases
- 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

These two scenarios should be avoided, as they can be confusing.

9.2.1 Command-line options for type-checking

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

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

9.3 Static Checking and Verification

This section will be added later.

9.3.1 Options specific to static checking

- **-prover *prover***: the name of the prover to use: one of z3_4_3, yices2 [TBD: expand list]
- **-exec *path***: the path to the executable corresponding to the given prover
- **-boogie**: enables using boogie (-prover option ignored; -exec must specify the Z3 executable)
- **-method *methodlist***: a comma-separated list of method names to check (default is all methods in all listed classes) [TBD - describe wildcards and fully
- **-exclude *methodlist***: a comma-separated list of method names to exclude from checking

- **-checkFeasibility *where***: checks feasibility of the program at various points: one of `none`, `all`, `exit` [TBD, 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

9.4 Runtime Assertion Checking

This section will be added later.

9.4.1 Options specific to runtime checking

- **-showNotExecutable**: warns about the use of features that are not executable (and thus ignored)
- **-racShowSource**: includes source location in RAC warning messages [TBD: default?]
- **-racCheckAssumptions**: enables (default on [TBD - is this default correct?]) checking `assume` statements as if they were asserts
- **-racJavaChecks**: enables (default on) performing JML checking of violated Java features (which will just proceed to throw an exception anyway)
- **-racCompileToJavaAssert**: (default off) compile RAC checks using Java asserts (which must then be enabled using `-ea`), instead of using `org.jmlspecs.utils.JmlAssertionFailure`
- **-racPreconditionEntry**: (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?]

9.5 Generating Documentation

This section will be added later.

9.6 Generating Specification File Skeletons

This section will be added later.

9.7 Generating Test Cases

This section will be added later.

9.8 Limitations of OpenJML's implementation of JML

Currently OpenJML does not completely implement JML. The differences are explained in the following subsections.

9.8.1 model import statement

OpenJML 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

9.8.2 purity checks and system library annotations

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.

9.8.3 TBD - other unimplemented features

Chapter 10

Using OpenJML and OpenJDK within user programs

The OpenJML software is available as a library so that Java and JML programs can be manipulated within a user's program. The developer needs only to include the `openjml.jar` library on the classpath when compiling a program and to call methods through the public API as described in this chapter. The public API is implemented in the interface `org.jmlspecs.openjml.IAPI`; it provides the ability to

- perform compilation actions as would be executed on the command-line
- parse files or Strings containing Java and JML source code, producing parse trees
- print parse trees
- walk over parse trees to perform user-defined actions
- type-check parse trees (both Java and JML checking)
- perform static checking
- compile modules with run-time checks
- emit javadoc documentation with JML annotations

The sections of this chapter describe these actions and various concepts needed to perform them correctly.

CAUTION: OpenJML relies on parts of the OpenJDK software that are labeled as internal, non-public and subject to change. Correspondingly, some of the OpenJML API may change in the future. The definition of the API class is intended to provide a buffer against such changes. However, the names and functionality of OpenJDK classes (e.g., the `Context` class in the next section) could change.

List classes CAUTION #2: The OpenJDK software uses its own implementation of Lists, namely `com.sun.tools.javac.util.List`. It is a different implementation than `java.util.List`, with a different interface. Since one or the other may be in the list of imports, the use of `List` in the code may not clearly indicate which type of List is being used. Error messages are not always helpful here. Users should keep these two types of List in mind to avoid confusion.

Example source code The subsections that follow contain many source code examples. Small source code snippets are shown in in-line boxes like this:

```
// A Java comment
```

Larger examples are shown as full programs. These are followed by a box of text with a gray background that contains the output expected if the program is run (if the program is error-free) or compiled (if there are compilation errors). Here is a "Hello, world" example program:

All of these full-program example programs are working, tested examples. They are available in the `demos` directory of the OpenJML source code. The opening comment line (as well as the class name) of the example text gives the file name.

The full programs presume an appropriate environment. In particular, they expect the following

- the current working directory is the `demos` directory of the OpenJML source distribution
- the Java `CLASSPATH` contains the current directory and a release version of the OpenJML library (`openjml.jar`). For example, if the demos directory is the current working directory and a copy of `openjml.jar` is in the `demos` directory, then the `CLASSPATH` could be set as "`.;openjml.jar`" (using the ; on Windows, a : on Mac and Linux)

Note that the examples often use other files that are in subdirectories of the `demos` directory.

```
// bash commands to compile and run the DemoHelloWorld example
  cd OpenJML/demos                     # Alter this to match your local installation
  export CLASSPATH=".;openjml.jar"     # Use a :  instead of ; on Unix or Mac
                                       # Copy openjml.jar to the demo directory
  javac DemoHelloWorld.java            # Be sure java tools from a 1.7 JDK
  java DemoHelloWorld                  # are on the PATH
```

10.1 Concepts

10.1.1 Compilation Contexts

All parsing and compilation activities within OpenJML are performed with respect to a *compilation context*, implemented in the code as a `com.sun.tools.javac.util.Context` object. There can be more than one Context at a given time, though this is rare. A context holds all of the symbol tables and cached values that represent the source code created in that context.

There is little need for the user to create or manipulate Contexts. However it is essential that items created in one Context not be used in another context. There is no check for such misuse, but the subsequent actions are likely to fail. For example, a Context contains interned versions of the names of source code identifiers (as `Names`). Consequently an identifier parsed in one Context will appear different than an identifier parsed in another Context, even if they have the same textual name. Do not try to reuse parse trees or other objects created in one Context in another Context.

Each instance of the `IAPI` interface creates its own Context object and most methods on that `IAPI` instance operate with respect to that Context. The `API.close` operation releases the Context object, allowing the garbage collector to reclaim space. [1]

10.1.2 JavaFileObjects

OpenJDK works with source files using `JavaFileObject` objects. This class abstracts the behavior of ordinary source files. Recall that the definition of the Java language allows source material to be held in containers other than ordinary files on disk; The `JavaFileObject` class accommodates such implementations.

OpenJML currently handles source material in ordinary files and source material expressed as `String` objects and contained in mock-file objects. Such mock objects make it easier to create source material programatically, without having to create temporary files on disk.

Although the basic input unit to OpenJDK and OpenJML is a JavaFileObject, for convenience, methods that require source material as input have variations allowing the inputs to be expressed as names of files or `File` objects. If needed, the following

[1] The OpenJDK software was designed as a command-line tool, in which all memory is reclaimed when the process exits. Although in principle memory can be garbage collected when no more references to a Context or its consitutent parts exist, the degree to which this is the case has not been tested.

methods create JavaFileObjects:

```
String filename = ...
File file = new java.io.File(filename);
IAPI m = Factory.makeAPI();
JavaFileObject jfo1 = m.makeJFOfromFilename(filename);
JavaFileObject jfo2 = m.makeJFOfromFile(file);
JavaFileObject jfo3 = m.makeJFOfromString(filename,contents);
```

The last of the methods above, `makeJFOfromString`, creates a mock-file object with the given contents (a String). The `contents` argument is a String holding the text that would be in a compilation unit. The mock-object must have a sensible filename as well. In particular, the given filename should match the package and class name as given in the `contents` argument. In addition to creating the `JavaFileObject` object, the mock-file is also added to an internal database of source mock-files; if a mock-file has a filename that would be on the source path (were it a concrete file), then the mock-file is used as if it were a real file in an OpenJML compilation. [TODO: Test this. Also, how to remove such files from the internal database.]

10.1.3 Interfaces and concrete classes

A design meant to be extended should preferably be expressed as Java interfaces; if client code uses the interface and not the underlying concrete classes, then reimplementing functionality with new classes is straightforward. The OpenJDK architecture uses interfaces in some places, but often it is the concrete classes that must be extended.

Table 10.1 lists important interfaces, the corresponding OpenJDK concrete class, and the OpenJML replacement.

TODO: Add Parser, Scanner, other tools, JCTree nodes, JMLTree nodes, Option/JmlOption, DiagnosticPosition, Tool, OptionCHecker

10.1.4 Object Factories

10.1.5 Abstract Syntax Trees

10.1.6 Compilation Phases and The tool registry

Compilation in the OpenJDK compiler proceeds in a number of phases. Each phase is implemented by a specific tool. OpenJDK examples are the `DocCommentScanner`, `EndPosParser`, `Flow`, performing scanning, parsing and flow checks respectively; the OpenJML counterparts are `JmlScanner`, `JmlParser`, and `JmlFlow`.

In each compilation context there is one instance of each tool, registered with the context. The Context contains a map of keys to the singleton instance of the tool (or its

Interface	OpenJDK class	OpenJML class
IAPI		API
	com.sun.tools.javac.main.Main	org.jmlspecs.openjml.Main
	Option	
IOption		JmlOption
IVisitor		
IJmlTree		
IJmlVisitor		
IProver		
IProverResult		ProverResult
IProverResult.ICounterexample		Counterexample
IProverResult.ICoreIds		
JCDiagnostic.DiagnosticPosition	SimpleDiagnosticPosition	DiagnosticPositionSE, DiagnosticPositi
Diagnostic<T>	JCDiagnostic	
	com.sun.tools.javac.main.JavaCompiler	JmlCompiler

Table 10.1: Interfaces and Classes

factory) for that context. The scanner and parser are treated slightly differently: there is a singleton instance of a scanner factory and a parser factory, but a new instance of the scanner and the parser are created for each compilation unit compiled. Tables 10.2 and 10.3 list the tools most likely to be encounterded when programming with OpenJML.

OpenJML implements alternate versions of many of the OpenJDK tools. The OpenJML versions are derived from the OpenJDK versions and are registered in the context in place of the OpenJDK versions. In that way, anywhere in the software that a tool is obtained (using the syntax `ZZZ.instance(context)` for a tool `ZZZ`), the appropriate version and instance of the tool is produced.

In some cases, a *tool factory* is registered instead of a tool instance. Then a tool instance is created on the first request for an instance of the tool. The reason for this is the following. Most tools use other tools and, for efficiency, request instances of those tools in their constructors. Circular dependencies can easily arise among these tool dependencies. Using factories helps mitigate this, though the problem still does easily arise.

TBD: Others - MemberEnter, JmlMemberEnter, JmlRac, JmlCheck, Infer, Types, Options, Lint, Source, JavacMessages, DiagnosticListener, JavaFileManager/JavacFileManager, ClassReader/javadocClassReader, JavadocEnter, DocEnv/DocEnvJml, BasicBlocker, ProgressReporter?, ClassReader, ClassWriter, Todo, Annotate, Types, TaskListener, JavacTaskImpl, JavacTrees

TBD: Others - JmlSpecs, Utils, Nowarns, JmlTranslator, Dependencies

Purpose	Java and JML tool	Notes
overall compiler	JavaCompiler, JmlCompiler	controls the flow of compilation phases
scanner factory	ScannerFactory, JmlScanner.Factory	
Token scanning	DocCommentScanner, JmlScanner	new instance created from the factory for each compilation unit
parser factory	ParserFactory, JmlFactory	
parser	EndPosParser, JmlParser	new instance created from the factory for each compilation unit
symbol table construction	Enter, JmlEnter	
annotation processing	Annotate	performed in JavaCompiler.processAnnotations
type determination and checking	Attr, JmlAttr	
flow-sensitive checks	Flow, JmlFlow	simple type-checking stops here
static checking	JmlEsc	invoked instead of desugaring if static checking is enabled (and processing ends here)
runtime assertion checking	JmlRac	invoked if RAC is enabled, and then proceeds with the remainder of compilation and code generation
desugaring generics		performed in the method JavaCompiler.desugar
code generation	Gen	not used for ESC

Table 10.2: Compilation phases and corresponding tools as implemented in JavaCompiler and JmlCompiler

Purpose	Java and JML tool	Notes
identifier table	Names	
symbol table	SymTab	
compiler and command-line options	Options, JmlOptions	
AST node factory	JCTree.Factory, JmlTree.Maker	
message reporting	Log	
printing ASTs	Pretty, JmlPretty	
name resolution	Resolve, JmlResolver	
AST utilities	TreeInfo, JmlTreeInfo	
type checks	Check, JmlCheck	
creating diagnostic message objects	JCDiagnostic.Factory	

Table 10.3: Some of the other registered tools

TBD: Is JmlTreeInfo still used

10.2 OpenJML operations

10.2.1 Methods equivalent to command-line operations

The `execute` methods of `IAPI` perform the same operation as a command on the command-line. These methods are different than others of `IAPI` in that they create and use their own `Context` object, ignoring that of the calling `IAPI` object.

The simple method is shown here:

```
import org.jmlspecs.openjml.IAPI;

IAPI m = new org.jmlspecs.openjml.API();
int returnCode = m.execute("-check","-noPurityCheck","src/demo/Err.java");
```

Each argument that would appear on the command-line is a separate argument to `execute`. All informational and diagnostic output is sent to `System.out`. The value returned by `execute` is the same as the exit code returned by the equivalent command-line operation. The String arguments are a varargs list, so they can be provided to `execute` as a single array:

```
import org.jmlspecs.openjml.IAPI;
String[] args = new String[]{"-check","-noPurityCheck","src/demo/Err.java"};
IAPI m = new org.jmlspecs.openjml.API();
int returnCode = m.execute(args);
```

A full example of using execute on a file with a syntax error is shown below:

□ ▯

A longer form of `execute` takes two additional arguments: a `Writer` and a `DiagnosticListener`. The `Writer` receives all the informational output. The `report` method of the `DiagnosticListener` is called for each warning or error diagnostic generated by OpenJML. Here is a full example of this method:

10.2.2 Parsing

There are two varieties of parsing. The first parses an individual Java or specification file, producing an AST that represents that source file. The second parses both a Java file and its specification file, if there is a separate one. The second form is generally more useful, since the specification file is found automatically. However, if the parse trees are being constructed programmatically, it may be useful to parse the files individually and then manually associate them.

Parsing constructs a parse tree. No symbols are created or entered into a symbol table. Nor is any type-checking performed. The only global effect is that identifiers are interned in the `Names` table, which is specific to the compilation context. Thus the only effect of discarding a parse tree is that there may be orphaned (no longer used) names in the `Names` table. The `Names` table cannot be cleared without the risk of dangling identifiers in parse trees.

Other than this consideration, parse trees can be created, manipulated, edited and discarded. Section TBD describes tools for manually creating parse trees and walking over them. Once a parse tree is type-checked, it should be considered immutable.

Parsing individual files

There are two methods for parsing an individual file. The basic method takes a `JavaFileObject` as input and produces an AST. The convenience method takes a filename as input and produces an AST. The methods of section 10.1.1 enable you to produce `JavaFileObjects` from filenames, File objects, or Strings that hold the equivalent of the contents of a file (a compilation unit).

```
JmlCompilationUnit parseSingleFile(String filename);
JmlCompilationUnit parseSingleFile(JavaFileObject jfo);
```

The filename is relative to the current working directory.

Here is a full example that shows both interfaces and shows how to attach a specification parse tree to its Java parse tree.

Parsing Java and JML files together

The more common action is to parse a Java file and its specification at the same time. The JML language defines how the specification file is found for a given source or binary class. In short, the specification file has syntax very similar to a Java file:

- it must be in the same package and have the same class name as the Java class
- if both are files, the filenames without suffix must be the same
- the specification file must be on the *specspath*
- if a .jml file meeting the above criteria is found anywhere on the specspath, it is used; otherwise a .java file on the specspath meeting the above criteria is used; otherwise only default specifications are used.[2]

Note that a Java file can be specified on the command-line that is not on the specspath. In that case (if there is no .jml file) no specification file will be found, although the user may expect that the Java file itself may serve as its own specifications. This is a confusing situation and should be avoided.

10.2.3 Type-checking

10.2.4 Static checking

10.2.5 Compiling run-time checks

10.2.6 Creating JML-enhanced documentation

10.3 Working with ASTs

10.3.1 Printing parse trees

TBD

[2]In the past, JML allowed multiple specification files and defined an ordering and rules for combining the specifications contained in them. The JML has been simplified to allow just one specification file, just one suffix (.jml), and no combining of specifications from a .jml and a .java file if both exist.

10.3.2 Source location information

TBD

10.3.3 Exploring parse trees with Visitors

OpenJML defines some Visitor classes that can be extended to implement user-defined functionality while traversing a parse tree. The basic class is `JmlScanner`. An unmodified instance of `JmlScanner` will traverse a parse tree without performing any actions.

There are three modes of traversing an AST.

- AST_JAVA_MODE - traverses only the Java portion of an AST, ignoring any JML annotations
- AST_JML_MODE - traverses the Java and JML syntax that was part of the original source file
- AST_SPEC_MODE - traverses the Java syntax and its specifications (whether they came from the same source file or a different one). This mode is only available after the AST has been type-checked.

A derived class can affect the behavior of the visitor in two ways:

- By overriding the `scan` method, an action can be performed at every node of an AST
- By overriding specific `visit...` methods, an action can be performed that is specific to the nodes of the corresponding type

In the example that follows, the scan method of the Visitor is modified to print the node type and count all nodes in the AST, the visitBinary method is modified to count Java binary operations, and the visitJmlBinary method is modified to count JML binary operations. The default constructor of the parent Visitor class sets the traversal mode to AST_JML_MODE.

The second example shows the differences among the three traversal modes. Note that the AST_SPEC_MODE traversal fails when requested prior to type-checking the AST.

There are two other points to make about these examples.

- Note that each derived method calls the superclass version of the method that it overrides. The superclass method implements the logic to traverse all the children of the AST node. If the super call is omitted, no traversal of the children is performed. If the derived class wishes to traverse only some of the children, a specialized implementation of the method will need to be created. It is easiest to create such an implementation by consulting the code in the super class.
- In the examples above, you can see that the System.out.println statement that prints the node's class occurs before the super call. The result is a pre-order traversal of the tree; if the print statement occurred after the super call, the output would show a post-order traversal.

10.3.4 Creating parse trees

10.4 Working with JML specifications

10.5 Utilities

– version – context – symbols

Chapter 11

Extending or modifying JML

JML is modified by providing new implementations of key classes, typically by derivation from those that are part of OpenJML. In fact, OpenJML extends many of the OpenJDK classes to incorporate JML functionality into the OpenJDK Java compiler.

11.0.1 Adding new command-line options

11.0.2 Altering IAPI

11.0.3 Changing the Scanner

11.0.4 Enhancing the parser

11.0.5 Adding new modifiers and annotations

11.0.6 Adding new AST nodes

11.0.7 Modifying a compiler phase

Chapter 12

Contributing to OpenJML

12.1 GitHub

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

- The source code repositories
- A wiki describing how to create and use a development environment for OpenJML
- The issue reporting tool for recording and commenting on bugs or desired features

12.2 Organization of OpenJML source code

The material comprising OpenJML is found in the following interrelated repositories on GitHub.

- OpenJML: contains the core software for OpenJML, including the modified OpenJDK, the tests and tutorial demos for OpenJML, and the source code for the Eclipse plugin 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

CHeck this - are we going to break out the update site?

12.3 Creating a development environment

This is all on the wiki??

Eclipse materials are organized into *projects* and *workspaces*. Eclipse provides the commands to create cloned GitHub repositories directly in an Eclipse workspace. We prefer creating the cloned git repositories and working copies separate from the workspace for two reasons: so that it is easy to also perform command-line edits and git commands in the working copy; and so that new workspaces can be created that point to the same git working copy if the first workspace becomes corrupted (as has occasionally happened).

The following instructions are current as of this writing. The OpenJML project wiki on GitHub will contain any updates to this information.

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

```
git clone https://github.com/OpenJML/OpenJML.git
git clone https://github.com/OpenJML/JMLAnnotations.git
git clone https://github.com/OpenJML/OpenJMLDemo.git
git clone https://github.com/OpenJML/Specs.git
git clone https://github.com/OpenJML/OpenJML-UpdateSite.git
```

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

- JmlAnnotations — the source for the JML annotations library
- OpenJML/OpenJDK — the modified source of OpenJDK
- OpenJML/OpenJML — the source for the command-line OpenJML
- OpenJML/OpenJMLUI — the source for the OpenJML Eclipse plugin
- OpenJML/OpenJMLTests — the command-line unit and functionality tests for OpenJML
- OpenJML/OpenJMLGUITests — the RCPTT-based tests of the OpenJML plugin
- OpenJML/OpenJMLFeature — the Eclipse plugin feature definition
- OpenJML-UpdateSite — the Eclipse update site
- OpenJML/vendor — the vendor branch holding a pristine version of the OpenJDK code
- OpenJMLDemo — holds material for public demos, including the examples used in this book
- Specs — the JML specifications of the Java system libraries
- OpenJML-UpdateSite — staging for the Eclipse update site

Then follow these instructions to create the Eclipse projects:

- You must also have Java 8 installed.
- Then launch Eclipse (a version at least as recent as Neon) and choose some new location as a Workspace location.
- Open Eclipse's *File » Import » General » Existing Projects into Workspace* wizard.
- Select $WC as the root directory.

- All of the items listed in the directory structure above should be listed (and selected) as available projects.
- TBD FINISH

12.4 Development of OpenJML

The source programming language for OpenJML is Java. The development environment of choice is Eclipse. The procedures described here are for that environment. Any future Developers wishing to contribute to OpenJML can retrieve a project-set file to download source code from GitHub and create the corresponding projects within Eclipse from `http://jmlspecs.sourceforge.net/OpenJML-projectSet.psf`. <p>Alternately, the set of SVN commands needed to checkout all the pieces of the OpenJML source code into the directory structure expected by Eclipse is found at this link: `http://jmlspecs.sourceforge.net/svn_commands`.

The general instructions for setting up a development environment are found at the JML wiki: `https://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJmlSetup`.

12.5 JML

Move this elsewhere or delete

The Java Modeling Language (JML) is a language that enables logical assertions to be made about Java programs. The assertions are expressed as structured Java comments or Java annotations. Various tools can then read the JML information and do static checking, runtime checking, display for documentation, or other useful tasks.

More information about JML can be found on the JML web site: `http://www.jmlspecs.org`. The information includes publications, a list of groups using or contributing to JML, mailing lists, etc. There is also a SourceForge project for JML : `https://sourceforge.net/projects/jmlspecs/`.

12.6 OpenJDK

Move this elsewhere or delete

OpenJDK (`http://openjdk.net`) is the project that produces the Java JDK and JRE releases. OpenJML extends OpenJDK to produce the OpenJML tools. OpenJML is a fully encapsulated, stand-alone tool, so the OpenJDK foundation is only of interest to OpenJML developers. Users, however, can be assured that OpenJML is built on 'official' Java tooling and can readily stay up to date with changes in the Java language.

TODOs

- Fix the TITLE for the web pages
- on HTML pages boxed examples do not render correctly

Bibliography

Index