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 March 26, 2018

Contents

Foreword

Gary write this?

Preface

General background: purpose, limitations, scope, acknowledgments

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

- This document is not a reference manual for JML. That document can be found at TBD .

- This document describes only OpenJML, not other tools implementing JML, such as

    - the KeY tool — `https://www.key-project.org/` — including a book about KeY: `https://www.key-project.org/thebook2/`

  However, the tutorial on writing specifications is broadly applicable.

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 containing information about 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

- `FIXME` is the most current version of this document

- The source code for OpenJML is stored in the github project at `http://www.github.com/OpenJML/OpenJML`

- the original JML tools and some other older (typically obsolete and no longer maintained) JML projects are contained in the jmlspecs github project at `http://sourceforge.net/projects/jmlspecs`.

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

- The KeY tool - `http://www.key-project.org/`

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

- Other tools and projects listed at `jmlspecs.org`.

Add google groups   Add the web site to try OpenJML

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

- jmlspecs-interest - general discussions about JML - `https://lists.sourceforge.net/lists/listinfo/jmlspecs-interest`

- jmlspecs-developers - news about active development in JML - `https://lists.sourceforge.net/lists/listinfo/jmlspecs-developers`

- jmlspecs-releases - news about releases of JML-related tools - `https://lists.sourceforge.net/lists/listinfo/jmlspecs-releases`

- You can watch activity on OpenJML by adding yourself to the watch list at `https://github.com/OpenJML/OpenJML`. There are companion projects that you may also want to watch, shown on `https://github.com/OpenJML`.

- Other less active mailing lists are listed here: `https://sourceforge.net/p/jmlspecs/mailman/`

If these sourceforge-based mail groups are inactive in the future, check for a corresponding project on GitHub.

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

Chapter 3

Quick start to OpenJML

An installation of the tool is needed to work through the tutorial. Follow the procedure in Appendix A or Appendix B to install OpenJML. If you are using the command-line tool, then all the installation files will be in a folder of your choice, which we designate as *$OJ*.

As described in the appendices you need

- an installation of Java 8

- an installation of Eclipse Neon.1 or later, if you are using the Eclipse GUI

- the OpenJML software itself

- an SMT solver

3.1   The tutorial examples

All the examples in this tutorial and in the *OpenJML User's Guide* are contained in the tutorial zip file so you as the reader can follow along, replicating the examples and modifying them to do your own exploration.

Download the file TBD and unzip it into a new folder of your choice, which we will designate by .

Fix the reference to *$TUT*

Say something about the organization and use of the tutorial files

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

3.2 Quick start to using OpenJML on the command-line

OpenJML is used as a typical command-line tool:

<div align="center">

java -jar *$OJ*/openjml.jar *<options> <files>*

</div>

Example commands will be shown throughout this tutorial. A full description of the options is given in Chapter **??**. Options and files can be intermixed. Options start with a single hyphen character; values of options can be given as the next command-line argument or, more clearly with an = character and value directly after the option, without white space, as in `-escMaxWarnings=3`. Files can be designated by absolute path or by a path relative to the current directory. If a path to a folder is given instead of a file, then all the `.java` files in the folder, recursively, are processed.

The OpenJML User Guide describes all the details of using OpenJML.

3.3 Quick start to using the OpenJML GUI

The OpenJML GUI is used consistent with using Eclipse for Java development. Open-JML is installed in Eclipse as a regular plug-in (cf. Appendix B). All the actions available to the user are given in submenu items of a top-level JML menu item, and are replicated in a few toolbar additions and in view menus and right-click context menus in various views.

The code that you wish to investigate with OpenJML is imported into Eclipse as a project. The following steps illustrate this process using the tutorial examples:

- Download the tutorial examples from HERE-TODO

- Unzip the material into a clean directory of your choosing, which we refer to as *$TUT*.

- In Eclipse, open the *File » Import...* wizard

- Choose *General » Projects from Folder or Archive*

- Click *Directory...* and navigate to the *$TUT* folder

- Select (with the checkbox) just the top-level folder

- Click *Finish* to import the folder into the Eclipse workspace as an Eclipse project

To operate on a file with OpenJML,

- Select the editor window with the file of interest

- Click the ESC item in the toolbar or invoke the Static Check (ESC) submenu item underneath the JML top-level menu item

- The file will be checked, with details being displayed in the JML console and any static warnings shown with conventional Eclipse problem markers in the editor.

- In various views (Package Explorer, Project Explorer, Navigator, Outline) you can select an individual method or a type or a folder or a whole project; you can also select multiple items. OpenJMLwill act on all of the relevant items within container objects, recursively.

- The coffee cup toolbar item performs just typechecking, the same as the `Typecheck` menu item.

- The `RAC` toolbar item compiles selected files for Runtime Assertion Checking (RAC).

Chapter 4

Some details

This chapter describes some basic preliminary details.

4.1 The form of JML annotations

The basic form of a JML annotation is a Java comment that begins with an symbol.
You can use either line or block comments, as in either

```
//@ requires arg != null;
//@ ensures \result > 0;
```
or
```
/*@
@ requires arg != null;
@ ensures \result > 0;
@*/
```

A clause (e.g., keyword + expression + semicolon) must be contained in a single JML
annotation comment; in the line comment form, that means the clause is contained in
one line.

The indentation
in the second
example is not
what is desired

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 tutorial and other documents, 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 now also used for Java annotations (a syntactic feature that cam well after JML began), the following problem can arise. Some Java code is written something like this (the particular Java annotation and its content are irrelevant):

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

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

JML annotations can be placed directly within .java files or in complementary .jml files. For compiled libraries where there are no source files, .jml files are required. The examples in this document generally show JML annotations within .java files. The user should be aware of the other mode and can read about it in the JML Reference Manual.

4.5 Modular reasoning

By design, static checking operates on a method at a time. Each method is considered in isolation: the method's implementation is compared to the method's specification using, where necessary, the specifications of other methods referred to in the implementation. The implementations of those other methods are not considered until it is their turn for checking. In this way each method can be checked on its own and the scale of the amount of code that needs consideration at once is limited to a single method, a design called *modular reasoning*.

In order for modular reasoning to be sound, every method must be proved individually to (a) satisfy its specifications and (b) terminate. In practice, this goal is only approximated: library specifications are often taken on faith and termination is often assumed. Nevertheless, each method proved consistent with its specifications is a step forward toward a more correct set of software.

Chapter 5

Pre- and Postconditions

As described in §4.5, the fundamental unit for checking software specifications is the method. Hence the first specifications to understand are method specifications In this chapter we will work through tutorial examples of 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 has become 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,

Listing 5.1: A countdown timer class

```java
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

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

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

Listing 5.3: Specifying the constructor

```java
//@ 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() {  ..    }
```

which is that any memory location at all (in the pre-state) may be modified. That 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

```java
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;

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
```

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

Figure 5.2: GUI output when checking demo1b/Timer
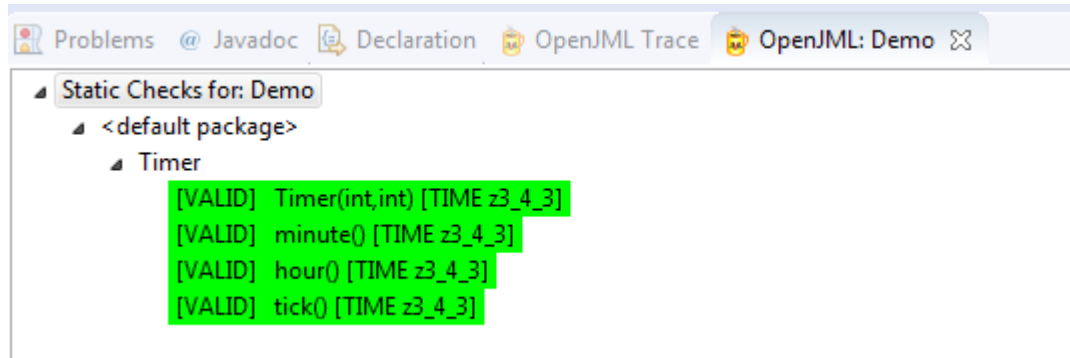
Listing 5.6: A test client for Timer

```java
public class Client {

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

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
```

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

Chapter 6

Introduction to OpenJML

6.1 Background on 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 is parsed through Java 8, as implemented in OpenJDK
- parse JML specifications for Java programs: all of JML is parsed, as described in this book
- typecheck all of JML: most of JML is checked, as described in this 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

Check 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 was 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 is 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 a companion document, the JML Reference Manual.

- §**??**: 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

6.2 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

- ...

6.3 License

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

The 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

OpenJML Concepts

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

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

25

classpath is specified by

<div align="center">-classpath *<path>*</div>

or

<div align="center">-cp *<path>*</div>

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 `-no-internalSpecs` 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.

The options to define these three paths are provided for the cases where they need to be carefully distinguished. However, in most situations it is sufficient to provide only an OpenJML classpath that includes all binary, source, ans specification files, letting it be used by default as the sourcepath and specspath as well.

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

The OpenJML tool is controlled by a variety of options, just as many other tools are. The implemented options are described in detail later (cf. §8.2); here we make a few general notes:

- Options for the command-line tool are expressed on the command-line, with single-hyphen prefixes, just as is the case for most other tools on Unix platforms. The most important consideration was to keep the style of options identical to that of OpenJDK.

- Options for the Eclipse GUI are chosen using typical Eclipse Preferences. There is a set of Preference pages for OpenJML. Most of the Preference items are a GUI version of a specific command-line option.

- Options interact with Java properties. Java properties can be used to set options without needing to state them on the command-line each time. This relationship is described below.

OpenJML uses Java properties to define values specified outside the command-line. Java properties are typical key-value pairs of two strings. Values for boolean options can be stated using the strings true and false. 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 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` file in the user's home directory (the value of the Java property `user.home`), if any
- A `openjml.properties` file in the current working directory (the value of the Java property `user.dir`), if any
- 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.

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

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

---

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

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

The properties that are currently recognized are these:

- `openjml.defaultProver` - the value is the name of the prover (cf. §7.3) to use
  by default
- `openjml.prover.`*name*, where *name* is the name of a prover, and the value is
  the file system path to the executable to be invoked for that prover (cf. §7.3)
- `org.jmlspecs.openjml.`*option*, where *option* is the name of an OpenJML option (without the leading hyphen)

TBD: Check the above

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 and personal configuration.
(If you are an OpenJML developer, take care not to commit your local `openjml.properties`
file into the OpenJML shared GitHub repository.)   Does the template file really have all of these?

7.3   SMT provers

The static checking capability of OpenJML uses SMT solvers to discharge proof obligations stemming from the specifications and implementation of a program. The SMT
solvers are not part of OpenJML itself. OpenJML needs to be told which solver to use
and the location of its executable in the local system. For the command-line tool, this
information is expressed in properties and command-line options; the GUI obtains this
information from the Preferences settings. For convenience, both the command-line
tool and the Eclipse plugin ship with a selection of SMT solvers and the tools are configured to use those by default. As part of setting up your installation of OpenJML,
you will need to obtain and install instances of any SMT solvers you want to use that
are not shipped with OpenJML; you also need to set the apporpriate properties or state
the name and location of the solver on the command-line.

SMT solvers in theory all read a common input format: the SMT-LIBv2.5 standard.[2]
OpenJML uses this standard and does not use extensions to it. However, solvers imperfectly implement this standard. So OpenJML uses a library, jSMTLIB[3], that translates

---

[2]`http://www.smt-lib.org`
[3]`http://smtlib.github.io/jSMTLIB/`

and delegates as needed. This means that OpenJML supports only (a) a fixed set of SMT solvers and (b) any solvers that are strictly compliant to SMT-LIBv2.5.

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

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

7.4 Conditional JML annotations

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

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

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

Each form relies on the *key* being defined or not. OpenJML defines keys using the `-keys` option, described in §**??**.

Should be able to define keys with properties

In OpenJML,

- the key `OPENJML` is enabled by default in the OpenJML tool

- the keys `ESC` and `RAC` are enabled when the respective OpenJML tools are being executed

- the key `DEBUG` is disabled by default; the **debug** statement (§**??**) is enabled if and only if the `DEBUG` is enabled

- the key `KEY` is reserved for the use of the KeY ([**?**]) tool and is disabled by default in OpenJML

- all other keys are disabled by default in OpenJML

7.5 Defaults for binary classes

TODO: Say more

7.6   Redundancy in JML and OpenJML

JML has a few features that explicitly allow redundancy.

TODO: Say more

7.7   Nullness and non-nullness of references

TODO: Say more

Chapter 8

Running OpenJML as a command-line tool

This chapter describes how to run and use the command-line version of OpenJML. You first need to install the software.

- The requirements and procedure for installing OpenJML are described in Appendix A.

- The Eclipse plug-in for OpenJML is described in §9; Appendix B contains the installation procedure for the plug-in.

8.1   Running OpenJML

8.1.1   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 (cf. the installation instructions in Appendix A).

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

Here *<options>* are command-line options in typical form (see §8.2) and *<files>* are 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 Tables 8.1 and 8.2, 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). Also the argument

to the `-cp` option will in general need to be a combination of the openjml jar file and other directories referenced by the Java files.

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

8.1.2 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
- 5 (`EXIT_CANCELLED`) : indicates exit because of user initiated cancellation (currently only within the GUI)

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.

Should add a special entry for static warnings

8.1.3 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) should be understood as a folder; all `.java` files recursively within the folder are included as if they were individually listed on the command-line. The `-dirs` option indicates that each one of the remaining command-line arguments is interpreted as either a source file (if it is a file with a `.java` suffix) or as a folder (if it is a folder) whose contents are processed as if listed on the command-line. Note that the `-dirs` option must be the last option. (These two options are necessary because otherwise the JDK will complain about the command-line arguments not being files.)

As described later in §8.1.4, 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?

8.1.4 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.]

8.1.5 Annotations and the runtime library

JML optionally uses Java annotations as introduced in Java 1.6. JML-defined annotation classes are in the package org.jmlspecs.annotation. In order for files using these annotations to be processed by Java, the annotation classes must be on the classpath (just like any other annotation classes). They are also 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 -no-internalRuntime. If you use this option, then you will have to supply your own annotation classes and (if using Runtime Assertion Checking) your own runtime utility classes on the classpath. You may wish to do this, for example, if you have newer versions of the annotation classes that you are experimenting with. You could simply put them on the classpath, since they would be in front of the

Options specific to JML	
–	no more options
-check	[8.2.2] typecheck only (`-command=check`)
-checkSpecsPath	[8.2.4] warn about non-existent specs path entries
-command <*action*>	[8.2.2] which action to do: check esc rac compile
-compile	[8.2.2] typecheck JML but just compile the Java code (`-command=check`)
-counterexample	[8.2.5] show a counterexample for failed static checks
-dir <*dir*>	[8.2.4] argument is a folder or file
-dirs	[8.2.4] remaining arguments are folders or files
-esc	[8.2.2] do static checking (`-command=esc`)
-internalRuntime	[8.2.4] add internal runtime library to classpath
-internalSpecs	[8.2.4] add internal specs library to specspath
-java	[8.2.2] use the native OpenJDK tool
-jml	[8.2.2] process JML constructs
-jmldebug	[8.2.7] very verbose output (includes -progress)
-jmlverbose	[8.2.7] JML-specific verbose output
-keys	[8.2.4] define keys for optional annotations
-method	
-nonnullByDefault	[8.2.4] values are not null by default
-normal	[8.2.7]
-nullableByDefault	[8.2.4] values may be null by default
-progress	[8.2.7]
-purityCheck	[8.2.4] check for purity
-quiet	[8.2.7] no informational output
-rac	[8.2.2] compile runtime assertion checks (`-command=rac`)
-racCheckAssumptions	[8.2.6] enables (default on) checking assume statements as if they were asserts
-racCompileToJavaAssert	[8.2.6] compile RAC checks using Java asserts
-racJavaChecks	[8.2.6] enables (default on) performing JML checking of violated Java features
-racShowSource	[8.2.6] includes source location in RAC warning messages
-showNotImplemented	warn if feature not implemented
-specspath	[8.2.4] location of specs files
-stopIfParseErrors	stop if there are any parse errors
-subexpressions	[8.2.5] show subexpression detail for failed static checks
-trace	[8.2.5] show a trace for failed static checks

Table 8.1: OpenJML options. See the text for more detail on each option.

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

automatically added classes and used in favor of default versions; however, if you want to be sure that the default versions are not present, use the `-no-internalRuntime` option.

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

8.2   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?

8.2.1   Command-line options

OpenJML's command-line options operate in a fashion similar 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 (§**??**). 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 non-required 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

8.2.1.1 Options: Operational modes

These operational modes are mutually exclusive.

- **-jml** (default) : use the OpenJML implementation to process the listed files, including embedded JML comments and any corresponding `.jml` files

- **-no-jml**: uses the OpenJML implementation to type-check and possibly compile the listed files, but ignores all JML annotations in those files

- **-java**: processes the command-line options and files using only OpenJDK functionality. No OpenJML functionality is invoked. Must be the first option and overrides the others.

8.2.2 Options: JML tools

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

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

- **-check** : causes OpenJML to do only type-checking of the Java and JML in the input files (alias for -command=check)

- **-compile** : causes OpenJML to do JML type-checking (as with -check), but then compiles the Java code without any runtime-checking (a rarely used option) (alias for -command=compile)

- **-esc** : causes OpenJML to do (type-checking and) static checking of the JML specifications against the implementations in the input files (alias for -command=esc)

- **-rac** : compiles the given Java files as OpenJDK would do, but with JML checks included for checking at runtime (alias for -command=rac)

- **-doc** : executes javadoc but adds JML specifications into the javadoc output files (alias for -command=doc) *Not yet implemented.*

8.2.3 The -no-internalSpecs option.

As described in §8.1.5, this option turns off the automatic adding of the internal specifications library to the specspath. If you use this option, it is your responsibility to provide an alternate specifications library for the standard Java class library. If you do not

you will likely see a large number of static checking warnings when you use Extended Static Checking to check the implementation code against the specifications.

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

8.2.4   Options: OpenJML options applicable to all OpenJML tools

- **-dir** <*folder*> : abbreviation for listing on the command-line all of the .java files in the given folder and its subfolders (recursively); if the argument is a file, use it as is. The argument may also be a path expression containing wild-cards (* and ?); such arguments are expanded into a list of files by Java programmatic glob expansion.

- **-dirs** : treat all subsequent command-line arguments as if each were the argument to -dir

- **-specspath** <*path*> : defines the specifications path, cf. section TBD

- **-keys** <*keys*> : the argument is a comma-separated list of JML keys (cf. the JML Reference Manual), used to conditionally enable selected annotations

- **-strictJML** : warns about any OpenJML extensions to standard JML

- **-showNotImplemented** : prints warnings about JML features that are ignored because they are not implemented; the default is disabled.

- **-nullableByDefault** : sets the global default to be that all declarations are implicitly @Nullable, if they are not explicitly declared @NonNull

- **-nonnullByDefault** : sets the global default to be that all declarations are implicitly @NonNull (the default), if not explicitly declared @Nullable

- **-purityCheck** : turns on (default is off) purity checking for library methods (currently -no-purityCheck is recommended since the Java library specifications are not complete for @Pure declarations)

- **-checkSpecsPath** : TODO

8.2.5   Options: Extended Static Checking

These options apply only when performing ESC:

- **-prover** <*prover*> : the name of the prover to use: one of z3_4_3, z3_4_5, cvc4, yices2

- **-exec** <*file*> : the path to the prover executable to use

- **-boogie** : enables using boogie (-prover option is ignored; -exec must specify the Z3 executable for Boogie to use) *Not yet implemented*

- **-method** *<methodlist>* : a semicolon-separated list of method names to check (default is all methods in all listed classes). In order to disambiguate methods with the same name, the items in the list may be fully-qualified method names, may include signatures (containing fully-qualified type names), and may be regular expressions (cf. §12.2.2)

- **-exclude** *<methodlist>* : a semicolon-separated list of method names to exclude from checking (default is to exclude none). The format for the items in the list is the same as for **-method** (cf. §12.2.2)

- **-checkFeasibility** *<where>* : checks feasibility of the program at various points — a comma-separated list of one of `none`, `all`, `exit`, `debug` [TBD, finish list, give default]

- **-escMaxWarnings** *<int>* : the maximum number of assertion violations to look for; the argument is either a positive integer or `All`; the default is `All`

- **-counterexample** : prints out a counterexample for failed proofs

- **-trace** : prints out a counterexample trace for each failed assert (includes `-counterexample`)

- **-subexpressions** : prints out a counterexample trace with model values for each subexpression (includes `-trace`)

8.2.6 Options: Runtime Assertion Checking

These options apply only when doing RAC:

- **-showNotExecutable** : warns about the use of features that are not executable (and thus ignored); turn off with `-no-showNotExecutable` What is the default

- **-showRacSource** : enables including source code information in RAC error messages (default is enabled; disable with `-no-showRacSource`)

- **-racCheckAssumptions** : enables checking `assume` statements as if they were asserts (default is enabled; disable with`-no-racCheckAssumptions`) Is this default correct?

- **-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`)

- **-racCompileToJavaAssert** : compile RAC checks using Java asserts (which must then be enabled using `-ea`) (default is disabled; disable with `-no-racCompileToJavaAssert`)

8.2.7   Options: JML Information and debugging

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

- **-help** : prints out help information about the command-line options

- **-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.

- **-quiet** : no informational output, only errors and warnings

- **-normal** : (default) some informational output, in addition to errors and warnings

- **-progress** : prints out summary information as individual files are processed and proofs are attempted (includes `-normal`)

- **-verbose** : prints out verbose information about the Java processing in OpenJDK (does not include other OpenJML information)

- **-jmlverbose** : prints out verbose information about the JML processing (includes `-verbose` and `-progress`)

- **-jmldebug** : prints out (voluminous) debugging information (includes `-jmlverbose`)

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

Other debugging options:

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

An option used primarily for testing:

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

8.2.8   Java Options: Version of Java language or class files

- **-source** *<level>* : this option specifies the Java version of the source files, with values of `1.4, 1.5, 1.6, 1.7, 1.8`... or 4, 5, 6, 7, 8, ... 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 library specification files are supported only for Java 1.8..

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

8.2.9 Java Options: Other Java compiler options applicable to Open-JML

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

These options control where output is written:

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

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

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

- **-cp** or **-classpath**: the parameter gives the Java classpath to use to find referenced classes whose source files are not on the command-line (cf. section TBD)
- **-sourcepath**: the parameter gives the sequence of directories in which to find source files of referenced classes that are not listed on the command-line (cf. section TBD)
- **-deprecation**: enables warnings about the use of deprecated features (applies to deprecated JML features as well)
- **-nowarn**: shuts off all compiler warnings, *including the static check warnings produced by ESC*
- **-Werror**: turns all warnings into errors, including compiler, JML type-checking and JML static check warnings
- **-verbose**: turn on Java verbose output (does not control JML output)
- **-Xprefer:source** or **-Xprefer:newer**: when both a .java and a .class file are present, whether to choose the .java (source) file or the file that has the more recent modification time [TBD - check that this works]
- **-stopIfParseErrors**: if enabled (disabled by default), processing stops after parsing if there are any parsing errors (TBD - check this, check the default)

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

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

8.2.10   Java options related to annotation processing

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

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

Chapter 9

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 GUI operates as follows: actions in the GUI prepare the equivalent of an OpenJML command-line (command, options, and files); that command is sent to the core OpenJML engine; problem reports, proof results, text, and other results from the engine are then displayed in various ways in the GUI. Options are defined by Eclipse preferences; the files that are acted on are determined by the current selection; the command to execute corresponds to the menu item, toolbar, or key-combination invoked.

9.1 GUI Features

9.1.1 Logo and icon

Note that the JML logo, shown to the left, is a JML-decorated yellow coffee cup; this logo in various sizes is used as an icon associated with various GUI elements. Various versions of the logo can be seen at `http://jmlspecs.org/logo.shtml`.

9.1.2 Selecting the target of commands

IMPORTANT: Java files in Eclipse that are to be acted on by OpenJML must be contained in an eclipse Java project.

Many commands operate on one or more methods, types, files, folders, packages or projects. The unit of operation is typically either a method or a file. When a command

is invoked, it operates on all the items that are *selected* in a View that has focus at the time it is invoked. If the item is a container for the units on which the command operates, then all the units in that container, recursively, are acted on sequentially. For example, the `Delete JML Markers` command operates on files; selecting a folder will apply the operation to every file, recursively, in that folder. Similarly, the `Static Check (ESC)` command operates on methods; selecting any type, file, folder or project will cause the operation to be performed on each method recursively within those containers.

Items are selected in the usual ways in Eclipse:

- Items can be selected in views that give a tree-structured organization by a single-click on an item, with shift-click or control-click used to extend selections to more than one item, such as the Package Explorer view, the Project Explorer view, the Navigator view, or the Outline view.

- Items can also be selected by giving focus to an editor pane, in which case just the one file being edited in that pane is selected.

- Some commands are available through (right-click) context menus in those views (on just the item clicked on) or the Problems View, custom OpenJML Views, or in Editor windows.

9.1.3 Commands

The OpenJML plug-in adds a number of commands to the IDE. 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 below, with forward references to more detailed discussion. The most common commands to understand and use are highlighted in bold and color. A few commands are also available on the toolbar (cf. §9.1.5).

Commands implemented in the OpenJML plug-in through menu and toolbar additions.

- 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. JML markers are ordinarily deleted automatically, but some-

times it is helpful to delete them manually in order to know that new markers are freshly generated.

- 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.

9.1.4   Menubar additions

The main Eclipse menubar contains an additional menu titled 'JML', circled in red in Fig. 9.1. It contains various submenu items, as shown in Fig. 9.2; the action for a menu item is the similarly named command, described in §9.1.3. Individual menu items may

Figure 9.1: JML menu item on the Eclipse menubar
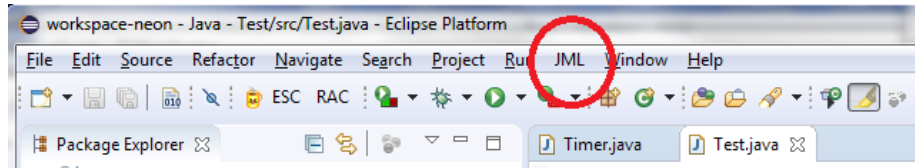
be disabled (grayed out) when they are not applicable. For instance, some items are enabled only when something is selected, some only when exactly one appropriate item is selected, some only when a method is selected, etc.

Figure 9.2: JML submenu items
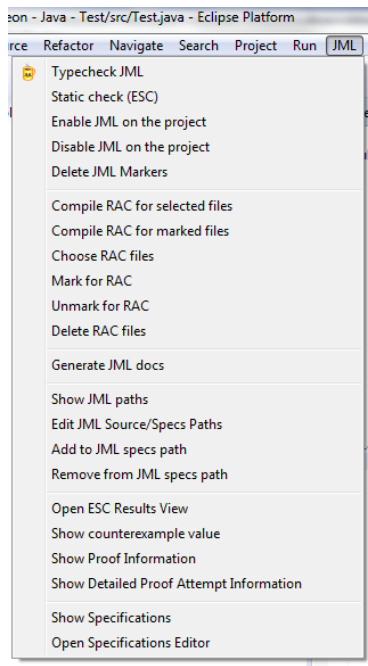
The menu items (and the corresponding toolbar items described in §9.1.5) act on the contents of whichever files, folders or projects are *selected*, as described in §9.1.2.

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... clarify
- ... OpenJML Proof view ... clarify

9.1.5 Toolbar additions

The OpenJML plug-in adds three toolbar items, circled in red in Fig. 9.3; clicking the toolbar item executes a corresponding command. These are the same operations as the corresponding items on the JML submenu and operate on the selected items as noted in §9.1.2.

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

Figure 9.3: JML additions to the Eclipse toolbar
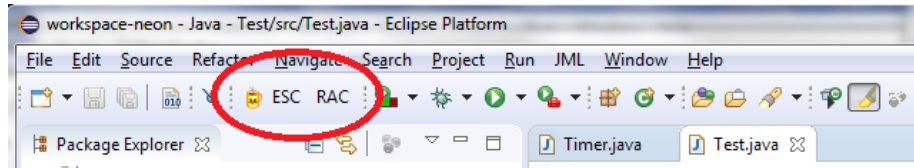
- RAC - executes the 'RAC - compile selected' command, described further in §**??**

9.1.6 OpenJML Problems, Markers and highlights

Needs revision for proper use of markers and annotations and icons

Eclipse uses *markers* to indicate the location of warnings and errors (generically, *problems*) in editor windows for 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 9.1.

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. 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 or not the icon is shown in the vertical ruler (along the left edge of the editor pane)

- whether or not a navigation mark is shown in the overview ruler (along the right edge of the editor pane)

- whether or not 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

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

The annotations in the lower part of Table 9.1 are used in showing counterexample paths (cf. §**??**). If supported by the underlying SMT solver, each static check warning

Table 9.1: 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

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.

Add icons into this table

Need some screen shots of markers and counterexample highlighting

9.1.7 OpenJML console and error log

The OpenJML plug-in adds an additional type of console. General informational output from the OpenJML engine 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. §9.1.6) 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 Console View's New Console menu has an additional item that creates a new OpenJML Console, as highlighted in Fig. 9.4. 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 may need to be brought to the front by hand.

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 share the

Figure 9.4: The JML Console
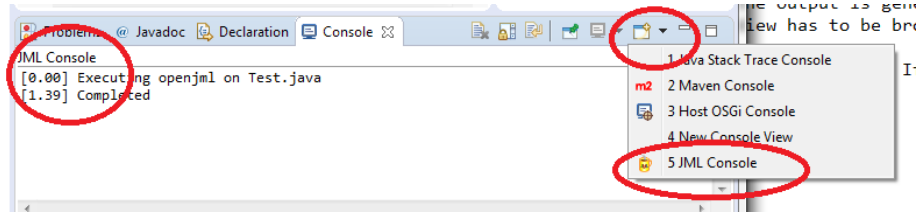
same instance of a JML Console.

9.1.8   Enabling automated checking

This section needs checking

The toolbar and menu commands allow the user to manually initiate type checking, static checking, or compiling with runtime assertion checks. These actions can also take place automatically whenever a file is automatically compiled. Automatic compilation happens when

- a file is saved and

- the 'Build Automatically' option (under the 'Project' menu, cf. Fig. 9.5) is enabled.

To have JML commands automatically performed whenever a Java file is compiled, the containing project must have the JML Builder enabled. This is accomplished by running the *Enable JML* menu (cf. Fig. 9.7) command on the project. As a result you will see

- a JML icon (the yellow coffee cup) decorating the project icon (cf. Fig. **??**) in, for example, the Package Explorer, and

- in the pane showing the Builders in the Project Properties, there will be a JML Builder shown enabled (cf. Fig. **??**).

The JML Nature can be disabled with the *Disable JML* command.
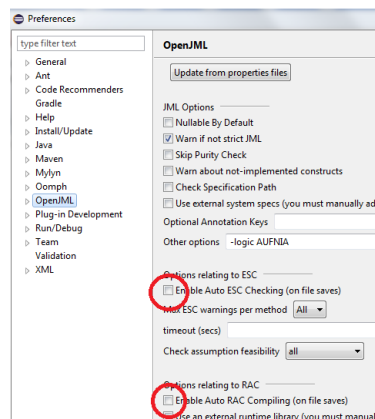
Figure 9.9: The OpenJML preferences for on-save actions

By default a JML-enabled project will execute type-checking automatically, showing markers for parsing and type errors. You can in addition enable static checking and runtime assertion compiling by enabling the corresponding options on the OpenJML Preferences page, as shown in Fig. 9.9. These options however, should be used with caution. If ESC is time-consuming on the files in the relevant project, it may not be desirable to initiate an ESC operation on every save.
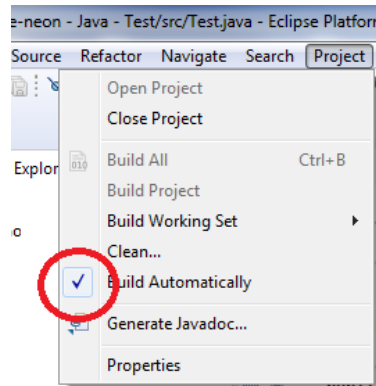
Figure 9.5: The Build Automatically option

Figure 9.6: The JML decorator marking a JML-enabled project

Figure 9.7: The Enable JML menu command
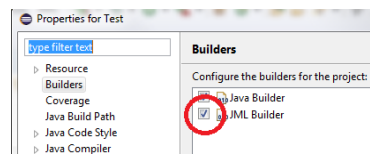
Figure 9.8: The project Builder list showing the JML Builder

As for runtime-assertion compiling, it will be performed on every save as well on whichever files have been edited and saved; the user may wish to be selective about which files are compiled with RAC assertions.

9.1.9 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:
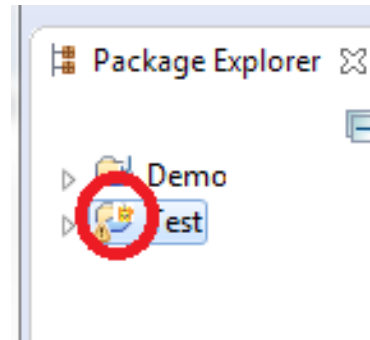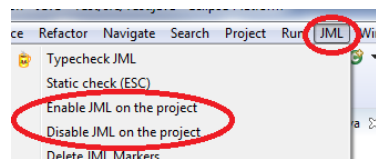
- A top-level page named 'OpenJML' found in the top-level list of Eclipse preference pages. This page allows setting options that would otherwise be set on the command-line. There is also a UI option, named 'UI verbose', that enables verbose output to the OpenJML console about actions within the UI code.
- A sub-page named 'ESC Job Control'. This page contains options controlling the granularity and parallelization of static checking operations. Need a section describing these options.
- A sub-page named 'Solvers'. (Click the turnstile next to 'OpenJML' in the list of Preference pages to see the subpage). These preferences allow specifying the executables for SMT solvers and setting the default solver to use.

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

9.1.10 Editor embellishments

TBD - fill out

- counterexample hovers

- quick fix proposals

- context-sensitive completions

- insertions

9.1.11 OpenJML Views

The OpenJML plug-in adds a few custom Views to Eclipse. These appear automatically when they are needed; they cannot be manually opened from the *Window»Show Views* menu.

proof summary, traces
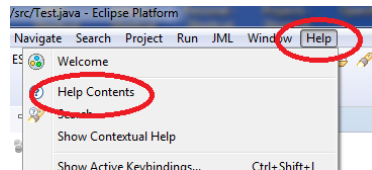
Figure 9.10: The Eclipse Help menu item
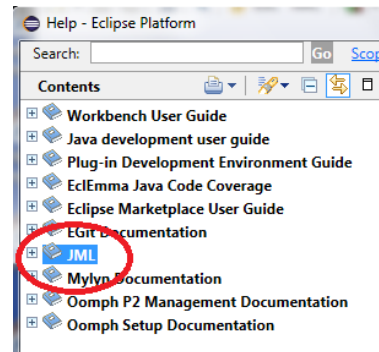
Figure 9.11: Eclipse Help table of contents

9.1.12   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. 9.10; then select 'JML' in the displayed Help table of contents as shown in Fig. 9.11 (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.

Check jml_orange.gif vs. esc_problem.gif; also JMLSmallDisabled

Add figure of JML help TOC and note the pdfs

9.1.13   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?

- quick fixes

- quick assists

- auto completion

- Internationalization. TBD...

- classpath intializer. TBD ...

- definition as an Eclipse project. TBD ...

- Open JML Perspective. TBD ...

Chapter 10

Classpaths, sourcepaths, and specification paths in OpenJML

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.
- 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.

Table 10.1: Location of JML Specs

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

- 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 re-
- Java class file on the classpath or in the Java system library (referred to by files compiled cases 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.

Chapter 11

OpenJML tools — Parsing and Type-checking

11.1 Type-checking JML specifications

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.

11.2 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 type declarations 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`

11.3   Automating type-checking

Parsing and typechecking JML is generally quite fast, so it is convenient to have such checking performed automatically. §**??** describes how to enable JML typechecking whenever a file is saved.

It is planned that typechecking be optionally performed *as-you-type*, much as the Java text is checked. However, this is not yet implemented.

Chapter 12

OpenJML tools — Static Checking (ESC) and Verification

Type-checking is performed automatically prior to ESC (Extended Static Checking). Thus ESC also depends on the information described in Chapters 8 and 11, particularly including the command-line options relevant to type-checking and the discussion of class, source, and specification paths in §7.1.

12.1   Results of the static checking tool

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

- It issues one or more static checking warnings.

- It finds no warnings through static checking and checks feasibility.

- It exhausts memory resources or alloted time.

- It encounters some internal bug.

These scenarios are discussed in the following subsections.

12.1.1   Finding static faults

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

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

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

Give an example

12.1.2   Checking feasiblity

A run of OpenJML with `-esc` may find no warnings through static checking. In this case, the tool runs additional checks to be sure the program is *feasible*, that is, that the specifications and the implementation actually permit execution of the program. By default, OpenJML will check that it is feasible to reach the beginning of the method body and to reach the exit point of the method. The beginning of the method body is not feasible if there is some contradiction within the preconditions and invariants.

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

Give an example

12.1.3   Timeouts and memory-outs

The underlying SMT solvers may report a time-out or memory exhaustion. One option is to increase the time out limit (with the `-timeout` option). An alternate recourse in this situation is to attempt to simplify the implementation or the specification. A time-out option to OpenJML is passed through to the underlying SMT solver for it to

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

Timeout used for each prover invocation

If OpenJML ends by exhausting memory, it is generally a problem with the solver. There is currently no control over the memory available to the SMT solver (aside from finding a larger computer). If it is Eclipse itself that exhausts memory, additional memory can be apportioned to Eclipse when it is launched by changing the Eclipse initialization parameters (in the eclipse.ini file in your installation of Eclipse).

12.1.4   Bugs

Despite the author's efforts, there still remain bugs in OpenJML. If you encounter any, please report them with as much information as possible, via the OpenJML project in GitHub. A useful bug report includes all the source code required to reproduce the problem, the operating system being used, the version of Java and OpenJML; the most useful reports will pare down the source code to a minimum amount that still provokes the error.

12.2   Options specific to static checking

12.2.1   Choosing the solver used to check

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

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

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

Table 12.1: Effect of `-method` and `-exclude`

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

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

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

12.2.2 Choosing what to check

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

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

The `-method` and `-exclude` options interact as shown in Table 12.1; in summary, `-exclude` overrides `-method`.

- If there are multiple instances of `-method` options, only the last one applies, as is the rule for all options. The same applies to the `-exclude` option. To specify multiple methods or exclude rules, use one option with a semicolon-separated list of strings.

- If a method is skipped because of these rules, then any classes or methods within the skipped method are also skipped.

- Despite the `-method` option, any method or type annotated with `@SkipEsc` is skipped

- The name of a constructor is the name of the class.

- There is no way to name anonymous classes or lambda functions in order to check or skip them.

- The list of strings to match is *semicolon*-separated rather than comma-separated because method signatures can contain commas. If multiple entires are separated by semicolons, you will likely have to quote the whole option to avoid the shell considering the semicolon the end of the command.

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

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

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

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

12.2.3   Choice of solver

These options specify which SMT solver is used to do static checking. Typically, however, the paths to executables are given in the `openjml.properties` file (cf. **??**). A default prover can also be specified in the `openjml.properties` file.

Need detail here

12.2.4   Detail about the proof result

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

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

Provide more information and examples

12.2.5 Controlling output

ESC can take a while to run if operating on a large set of software. It is useful then to have good progress reporting and to control the output produced. The basic controls are the level of verbosity, in particular the `-progress` setting and the options described in the previous subsection (§12.2.4).

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

- **-progress** : so that the starting and completing each method is reported; these delineations also serve to associate ewarning and error reports with the method that produced them

- **-escMaxWarnings=1** : just one warning per method saves time and is enough to tell whether further work will be needed. ALlow a higher limit when detailed analysis is being performed on just one or a few methods.

- **-checkFeasibility=exit** : in general the default value should be used to minimize computation time, but for an overarching run, just check feasibility of the exit point of the method to be sure the absence of warnings is not due to some contradictory requirements or axioms.

- Do not request tracing or counterexample information : this information is most helpful during debugging of single methods; in runs over many methods it just adds (voluminous) information that makes the output more difficult to understand

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

Note that when using the GUI, the OpenJML View gives a summary of proof results for each method.

Chapter 13

Runtime Assertion Checking

13.1 Compiling classes with assertions

Runtime-assertion checking (RAC) is accomplished by

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

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

- OpenJML is used is used instead of javac
- the option `-rac` is included
- the OpenJML runtime library (`jmlruntime.jar`) must be included in the class-path

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

There are a few points to note:

- Both openjml and javac will compile all the classes on the command-line and any classes referred to by those classes but not yet compiled. Hence it can be useful to perform a full javac compilation first, so no unexpected files have RAC enabled.
- Assertions are compiled only into classes compiled with `-rac`, and not into library classes or super classes.

- Assertion violations are reported only for the particular execution of the program. An absence of reports does not mean that some other run of the program (with different inputs) will be assertion-violation-free.

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

- well-definedness checks of any assertion or assumption, before the assertion or assumption itself is checked

- any explicit JML assert, reachable and unreachable statement

- any explicit JML assume statement (no checked by default)

- non-null checks when a object is deferenced (dot-operator or array-element operator)

- non-null checks when a reference variable or formal parameter declared NonNull is assigned

- array index is in range when an array is indexed

- checks implied by `assignable` clauses on any assignment

- checks implied by `accessible` clauses on any read

- pre-conditions and invariants of a callee, checked as assertions by the caller before calling a callee

- pre-conditions and invariants of a callee, checked as assumptions by a callee after being called but before executing the body of the callee (not checked by default)

- post-conditions and invariants of a callee, checked as assertions by a callee after executing the body of the callee

- post-conditions and invariants of a callee, checked as assumptions by a caller after returning from a callee (not checked by default)

- More? Label with the label that is used.

13.2 Options specific to runtime checking

13.2.1 -showNotExecutable

-showNotExecutable: (default:disabled) warns about the use of features that are not executable (and thus ignored). Some features of JML are not executable. If this op-

tion is enabled, warnings are printed during compilation when such features are used. Turning on this option can be helpful to a user unsure why a particular assertion is not being reported failing, just to be sure it is actually being compiled. The default is disabled.

13.2.2 -showNotImplemented

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

13.2.3 -racShowSource

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

As an example, the input file

```
public class A {

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

when compiled with the command
        `java -jar openjml.jar -rac -racShowSource A.java`
and run with
                    `java -cp ".;jmlruntime.jar" A`
produces the output

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

If compiled with
        `java -jar openjml.jar -rac -no-racShowSource A.java`
the output is

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

13.2.4 -racCheckAssumptions

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

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

So this input file

```
public class A {

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

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

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

when compiled with the command
        java -jar openjml.jar -rac -racCheckAssumptions A.java
and run with
                        java -cp ".;jmlruntime.jar" A

produces the output

```
A.java:4: JML precondition is false
    m(args.length);
     ^
A.java:8: Associated declaration: A.java:4:
  //@ requires i == 1;
       ^
A.java:8: JML precondition is false
  //@ requires i == 1;
       ^
A.java:16: JML postcondition is false
  public static int mm(int i) {
                     ^
A.java:15: Associated declaration: A.java:16:
  //@ ensures \result == 20;
       ^
A.java:5: JML postcondition is false
    mm(args.length);
     ^
A.java:15: Associated declaration: A.java:5:
  //@ ensures \result == 20;
        ^
```

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

However, if the example is compiled with

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

the output is

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

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

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

13.2.5 -racJavaChecks

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

The discussion in §13.2.7 below is also important to when and how JML violations are reported.

As an example, the input file

```
public class A {

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

}
```

when compiled with the command
```
        java -jar openjml.jar -rac -racJavaChecks A.java
```
and run with
```
                 java -cp ".;jmlruntime.jar" A
```
produces the output

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

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

If compiled with

        java -jar openjml.jar -rac -no-racJavaChecks A.java
the output is

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

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

13.2.6   -racCompileToJavaAssert

-racCompileToJavaAssert: (default: disabled) compiles RAC checks using Java asserts (which must then be enabled using -ea), instead of using org.jmlspecs.utils.JmlAssertionError. When this option is enabled, all assertion violation reporting is through Java assertion errors; that is, Option (C) in §13.2.7 is used despite any system properties. Furthermore, no reports will be generated at all at runtime unless the Java option -ea is enabled.

13.2.6.1   -racPreconditionEntry

-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?]

Need an example

13.2.7   Controlling how runtime assertion violations are reported

There are three ways in which a RAC-compiled program can report assertion violations. These can be controlled by properties set at the time the RAC-enabled program is *run* (not when it is *compiled*). Note that if the option -racCompileToJavaAssert is enabled (§13.2.6) then option (C) below is compiled in at compile time, and the various runtime alternatives described here are no longer available.

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

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

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

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

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

As an example, the input file

```
public class A {

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

when compiled with the command

java -jar openjml.jar -rac A.java

and run with

java -cp ".;jmlruntime.jar" A

produces the output

```
A.java:5: JML assertion is false
    //@ assert i == 1;
        ^
```

If compiled the same way but run with
`java -cp ".;jmlruntime.jar" -Dorg.jmlspecs.openjml.racshowstack A`
the output is

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

If compiled the same way but run with
`java -cp ".;jmlruntime.jar" -Dorg.jmlspecs.openjml.racexceptions A`
the output is

```
Exception in thread "main" org.jmlspecs.utils.JmlAssertionError: A.java:5: JML assertion is
    //@ assert i == 1;
        ^
at org.jmlspecs.utils.Utils.createException(Utils.java:99)
at org.jmlspecs.utils.Utils.assertionFailureL(Utils.java:52)
at A.main(A.java:1)
```

And if compiled the same way but run with
`java -cp ".;jmlruntime.jar" -ea -Dorg.jmlspecs.openjml.racjavaassert A`
the output is (Bad line numbers)

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

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

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

labels are listed **??** WHERE .

- If there is a system property `org.openjml.exception.`*`label`* defined for a
  given label, then the value of that property is expected to be the name of a class
  that is a subtype of `java.lang.Error`, and an exception of that class is thrown
  (if such an exception cannot be created, then an Error of type `org.jmlspecs.utils.JmlAssertionError`
  is thrown.

- If there is no such property defined, then an Error of type `org.jmlspecs.utils.JmlAssertionError$`*`label`*
  is thrown, if that type exists. Such a class is a nested class defined within
  `JmlAssertionError` and so must be part of the OpenJML runtime library. Cur-
  rently only `Precondition` and `PreconditionEntry` are defined, but others
  may be added in the future. All such nested classes are derived from `org.jmlspecs.utils.JmlAssertionError`

- If no such nested class is defined, then an Error of type `org.jmlspecs.utils.JmlAssertionError`
  is thrown.

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

13.2.8 RAC FAQs

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

13.2.8.1 Uncompiled fields and methods

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

Make an exmaple

13.2.8.2 Non-executable or unimplemented features

Some JML features are not executable by RAC. One example is a quantified expression
over unrestricted
`bigint` or
`real` variables. Also, some JML constructs are not implemented. If the OpenJML op-
tions are set so that no warnings are issued about non-executable or not-implemented

features, then some default value is used: expressions typically default to true and clauses typically default to being ignored. This can cause a difference in behavior between RAC and ESC and can also cause confusion in users when comparing RAC output to the JML specifications as written. The recommendation is to always enable the options `-showNotImplemented` and `-showNonExecutable` for any crucial or final or debugging runs of OpenJML.

Get and insert option names

Make example

13.2.8.3 Try blocks too large

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

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

Some tips to avoid this problem are these:

- Keep methods small

- Limit runtime assertions to just those needed to check crucial invariants and preconditions

- Use the `-no-racCheckAssumptions` option.

Chapter 14

Static and Runtime warnings

This chapter enumerates the kinds of warnings produced by Static and Runtime checking, with demonstrations of each kind of warning. For convenience, the warnings are listed in alphabetical order. However, a few of them are the most commonly occurring warnings.

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

Each warning is illustrated with an example. In each case the example is a class `Demo.class`. To run static checking on the example use this command, where *$OJ* is replaced by the path to the directory containing the OpenJML installation.

```
java -jar $OJ/openjml.jar -esc Demo.java
```

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

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

Then compile the `Demo class` with the command

```
java -jar $OJ/openjml.jar -rac Demo.java
```

and run it with the command (replacing the colon with a semicolon on Windows)

```
java -cp ".:$OJ/jmlruntime.jar" Demo
```

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

behaviors.

List the common ones? Brings the appendix here?

14.1 `ArithmeticCastRange` warning

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

```
public class Demo {

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

The result of ESC is

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

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

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

CAUTION: ignoring a warning still makes the assumption that the input was in range – check this out properly.   mention nowarn

14.2 `ArithmeticOperationRange` warning

The `ArithmeticOperationRange` warning is issued whenever an arithmetic operation cannot be assured to not cause an over or underflow. Note that over or underflow is

a property of the operation, not of any subsequent assignment of the intermediate value produced by the operation.

```
public class Demo {

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

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

The result of ESC is

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

The semantics of Java permits integer operations to overflow and wrap-around in 2's-complement arithmetic. So if intended, the operation is not illegal; however it can cause confusion. For instance, in Java, $(x+1) > (y+1)$ does not mean $x > y$, because y might be the maximum value of an `int`, and $y+1$ the minimum value.

If intended, the ESC warning can be suppressed using `nowarn` annotations.**??**

CAUTION: ignoring a warning still makes the assumption that the input was in range – check this out properly.   mention nowarn

14.3 `Assert` **warning**

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

```
public class Demo {

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

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

The result of ESC is

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

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

14.4 `Assume` **warning (RAC only)**

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

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

Thus this code

```
public class Demo {

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

compiled with

```
  java -jar $OJ/openjml.jar -rac -racCheckAssumptions Demo.java
```
and run with
```
              java -cp ".;$OJ/jmlruntime.jar" Demo 0
```
results in

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


Without the -racCheckAssumptions option, no output is emitted.

14.5 Constraint **warning**

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

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

```
public class Demo {

  private /*@ spec_public */ int count;

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

  //@ assignable count;
  public void increment() {
    count++;
  }

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

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

```
Demo.java:15: warning: The prover cannot establish an assertion (Constraint: Demo.java:5: )
    return count;
    ^
Demo.java:5: warning: Associated declaration: Demo.java:15:
  //@ public constraint count > \old(count);
            ^
2 warnings
```

14.6 `Initially` **warning**

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

The following example illustrates the combination of an `initially` clause and a de-

fault constructor:

```
public class Demo {

  public int count;

  //@ public initially count > 0;

}
```

The result of ESC on this example is

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

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

14.7 `ExceptionalPostcondition` **warning**

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

<div align="center">

signals (*Exc e*) *expr*;

</div>

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

$$(e \text{ instanceof } Exc) \rightarrow expr$$

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

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

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

```
public class Demo {

  static public int field;

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

  static void init() {
  }
}
```

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

14.8 `PossiblyNegativeIndex` warning

Array indices in array element access or assignment expressions must be non-negative values smaller than the size of the array. OpenJMLissues a `PossiblyNegativeIndex` warning if it cannot prove that the index of an array access or assignment expression is non-negative.

Applying ESC to this example

```
public class Demo {

  public static int[] arr;

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

results in this output:

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

14.9 `PossiblyNegativeSize` **warning**

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

int[] array = new int[x];

However, trying to create an array with a negative size will result in a runtime error (a
What? exception). OpenJMLissues a `PossiblyNegativeSize` warning if it cannot
prove that the argument of an array allocation expression is non-negative.

Applying ESC to this example

```
public class Demo {

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

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


}
```

results in this output:

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

14.10   `PossiblyTooLargeIndex` **warning**

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

Applying ESC to this example

```
public class Demo {

  public static int[] arr;

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

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

}
```

results in this output:

```

```

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

14.11 `Postcondition` **warning**

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

This example shows a situation in which the implicit non-null-ness of the return value is not established.

```
public class Demo {

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

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

The result of ESC is

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

14.12 `Precondition` **warning**

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

```
public class Demo {

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

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

The result of ESC is

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

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

14.13   `ExceptionalPostcondition` **warning**

14.14   `Assignable` **warning**

Chapter 15

Other OpenJML tools

15.1   Generating Documentation

This section will be added later.

15.2   Generating Specification File Skeletons

This section will be added later.

15.3   Generating Test Cases

This section will be added later.

15.4   Inferring specifications

This section will be added later.

Chapter 16

Limitations of OpenJML's implementation of JML

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

16.1   model import statement

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

In these two examples,

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

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

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

In

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

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

However,

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

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

TBD - more to be said about .jml files

16.2   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.

16.3   TBD - other unimplemented features

Chapter 17

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.

17.0.0.0.1   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.

17.0.0.0.2   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
```

17.1   Concepts

17.1.1   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]

17.1.2   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.]

17.1.3   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 17.1 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

17.1.4   Object Factories

17.1.5   Abstract Syntax Trees

17.1.6   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 17.1: 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 17.2 and 17.3 list the tools most likely to be encounterded when programming with OpenJML.

OpenJML implements alternate versions of many of the OpenJDK tools. The Open-JML 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 17.2: 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 17.3: Some of the other registered tools

TBD: Is JmlTreeInfo still used

17.2 OpenJML operations

17.2.1 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:

17.2.2   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.

17.2.2.1   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 17.1.1 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.

17.2.2.2 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.

17.2.3 Type-checking

17.2.4 Static checking

17.2.5 Compiling run-time checks

17.2.6 Creating JML-enhanced documentation

17.3 Working with ASTs

17.3.1 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.

17.3.2   Source location information

TBD

17.3.3   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 un-modified 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.
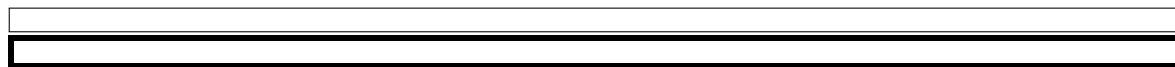
The second example shows the differences among the three traversal modes. Note that the AST_SPEC_MODE traversal fails when requested prior to type-checking the AST.
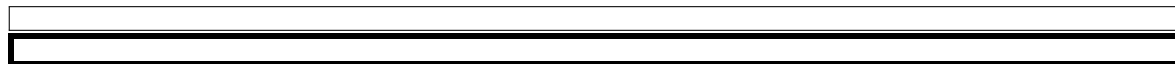
There are two other points to make about these examples.

- 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.

17.3.4 Creating parse trees

17.4 Working with JML specifications

17.5 Utilities

– version – context – symbols

Chapter 18

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.

18.1   Adding new command-line options

18.2   Altering IAPI

18.3   Changing the Scanner

18.4   Enhancing the parser

18.5   Adding new modifiers and annotations

18.6   Adding new AST nodes

18.7   Modifying a compiler phase

Chapter 19

Contributing to OpenJML

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

The source programming language for OpenJML is Java. The development environment of choice is Eclipse.

19.1 GitHub

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

- The OpenJML source code and related repositories
- A wiki describing how to create and use a development environment for Open-JML (`https://github.com/OpenJML/OpenJML/wiki`)
- The issue reporting tool for recording and commenting on bugs or desired features (`https://github.com/OpenJML/OpenJML/issues`)
- A number of other repositories that are related to JML, some of which are relevant to OpenJML.

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

- 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
- OpenJML-UpdateSite: the update site for the Eclipse plug-in

- Solvers: binary instances of SMT solvers
- SMT-Solvers: an Eclipse feature plug-in containing the Solvers project, so the solvers can be distributed through an update site
- openjml.github.io: the repository holding the material for the OpenJML website at `www.openjml.org`
- jdk8u-dev-langtools: the most recent snapshot of OpenJDK development merged into the OpenJDK folder in the OpenJML repository

In addition, Say more about these

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

19.2   Maintaining the development wiki

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

19.3   Issues

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

OpenJML does not yet use the project management features of GitHub. Is that going to change?

19.4   Creating a development environment

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 occasionally happens).

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
git clone https://github.com/OpenJML/Solvers.git
git clone https://github.com/OpenJML/SMT-Solvers.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/vendor — the vendor branch holding a pristine version of the Open-JDK code
- OpenJMLDemo — holds material for public demos, including the examples used in this book
- Specs — the JML specifications of the Java system libraries
- Solvers — binary executables of SMT solvers
- SMT-Solvers — Eclipse feature plugin for the solvers
- 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

You should set a variety of options to be consistent with other developers, as described in the wiki:

19.5   Running tests

To be written

19.6   Running a development version of the GUI

To be written

19.7   Building and testing releases

To be written

19.8   Packaging a release

To be written

19.9   Maintaining the project website

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

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

19.10   Updating to newer versions of OpenJDK

To be written

Appendix A

Installing OpenJML

A.1 System Requirements

Your system must have the following:

- A Java 1.8 JRE as described in section A.1.2. This must be the JRE in use in the environment in which OpenJML is invoked. OpenJML will not run with either Java 1.9 (yet) or version previous to Java 1.7; this is because OpenJML builds on OpenJDK, which at this point uses Java 1.8 features

- If tools needing SMT solvers are to be used you will need an SMT solver. Such tools include static checking, specification inference, and test generation. The installation of OpenJML includes some SMT solvers; in particular it includes Z3 4.3, which is the solver recommended to be used with OpenJML at present. Other options are given in §A.1.3 below, but are not well-tested or supported at present..

A.1.1 Operating System

OpenJML is regularly tested in MacOS X (10.12.6 - Sierra and 10.11.6 - El Capitan) environments; it is also run on Linux (Ubuntu); it has in the past been run on Windows 7, but there is not a current test environment on Windows, As the tool is a pure Java application, we expect it to work well in other environments also; however, the underlying SMT solvers may behave differently on different OSes or may not even be available. Feedback about success or failure in any environment is welcome.

A.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

The current release of OpenJML is not compatible with Java 1.9 or Java 1.7 or earlier versions. This restriction results from OpenJML being built on OpenJDK.

A.1.3 SMT solvers

You will need an SMT solver to perform static checking of JML specifications. Open-JML ships with a few solvers; in particular, at present OpenJML primarily supports and uses Z3 4.3, with Z3 4.5 in progress; Z3 4.4 is not as successful and is not recommended.

Note that the different SMT solvers have different performance and properties. For example, some handle some underlying theories (such as real arithmetic) better than others. So some solvers may time out or report an inability to prove something where others are successful. It is sometimes helpful to try more than one solver on a given problem.

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

A.2 Command-line tool download and installation

The OpenJML command line tool can be downloaded from

                http://jmlspecs.sourceforge.net/openjml.zip

or   https://github.com/OpenJML/OpenJML/releases/

The command-line tool is supplied as a .zip file. Download the file to a directory of your choice (referred to as *$OPENJML* subsequently) and unzip 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
- folders `Solvers-macos`, `Solvers-linux`, `Solvers-windows` containing SMT solvers for the respective platforms.
- `openjml-template.properties` - a sample file, which should be copied, renamed `openjml.properties`, and customized for your environment. It contains definitions of properties whose values depend on your local system or desired local environment. (cf. §A.3)

- `LICENSE.rtf` - a copy of the modified GPLv2 license that applies to OpenJDK and OpenJML
- `OpenJMLUserGuide.pdf` - this document

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

A.3   Local customization

OpenJML can be customized to your local environment as described in §7.2. Local properties are specified in a `openjml.properties` file, stored in the same directory as `openjml.jar` or in the user's home directory. The `openjml.properties` file can be used to indicate default command-line arguments and other local properties used by the tool. The installation includes the file `openjml.properties-template`, which can be copied and customized to create `openjml.properties`.

SMT solvers are needed if you intend to use the static checking capability of OpenJML (cf. §A.1.3). Recommended solvers are included in the installation package and are used by default. If you wish to use an alternate SMT solver, the location of the solver can be specified on the command-line or, more easily, in the `openjml.properties` file. For example, if the Z3 4.3 solver is located in your system at absolute location *<path>*, then include the following line in the `openjml.properties` file

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

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

Appendix B

Installing the OpenJML Eclipse plug-in

This chapter describes the system requirements and installation procedure for the Eclipse plug-in that encapsulates OpenJML. The details of using the plug-in are described in §9.

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

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

B.1 System Requirements

Your system must have the following:

- A Java 1.8 JRE as described in Appendix A. 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 4.6.3 (Neon) or later. (Development in Java 1.8 requires Neon or later.)

- One or more SMT solvers, if the static checking functionality will be used. Recommended solvers are shipped as part of the plug-in. See the list in §A.1.3.

B.2 Installation

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

- Start Eclipse in a Eclipse Workspace of your choosing

- 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 added to 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).

B.3 Local customization

The installation of the OpenJML Eclipse plug-in can be customized by specifying some local properties. For the most part, the settings that would be made using the command-line or `openjml.properties` files are set in the plug-ins Preference pages (cf. §**??**).

Describe how to use openjml.properties to initialize settings

Appendix C

Static warning categories

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

- Assertions or verification conditions generated by the semantics of Java and JML are reported by either ESC or RAC. These are listed in Table **??** Fix table - should be C.1

- Assumptions generated by the semantics of Java and JML are just assumed and not validated by ESC; RAC can optionally check them, under control of the option -racCheckAssumptions. These are listed in Table C.2.

- Some items are similarly named, beginning with either `Possibly`... or `Undefined`... . The `Possibly` label is used if the condition cannot be ruled out at the given location in Java code; the `Undefined`... label is used where the condition makes a JML expression not well-defined.

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

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

Table C.1b: Table C.1 continued.

Warning class	Description
`PossiblyBadCast`	
`PossiblyBadArrayAssignment`	assignment of a reference to an array where the reference type is not a subtype of the underlying array index type (a Java `ArrayStoreException`)
`PossiblyNullDeReference`	an expression being dereferenced is null
`PossiblyNullField`	a NonNull field has a null value when checked as part of invariants CHECK
`PossiblyNullValue`	the value for a switch, throw, or synchronized statement is null
`PossiblyNegativeSize`	the size of an array is negative
`PossiblyNegativeIndex`	the index of an array index operation is negative
`PossiblyToolargeIndex`	the index of an array index operation is larger or equal to the array length
`PossiblyPrecondition`	
`PossiblyNullUnbox`	a null reference is being unboxed to a primitive
`PossiblyNullAssignment`	a null value is being assigned to a NonNull location
`PossiblyNullInitialization`	a NonNull location is being initialized with a null value
`PossiblyDivideByZero`	the denominator of a division operation is 0
`PossiblyLargeShift`	the shift amount in a left shift operation is larger or equal to the number of bits in the left-hand argument (this is not illegal in Java, but usually surprises users)
`Postcondition`	a postcondition (`ensures` clause) is not valid
`Precondition`	reported when the composite precondition of a method called within the body of the method being checked cannot be proved valid
`Reachable`	
`Readable-if`	
`StaticInit`	

Table C.1c: Table C.1 continued.

Warning class	Description
`UndefinedBadCast`	
`UndefinedDivideByZero`	the denominator of a division operation is 0 in a JML expression
`UndefinedNegativeIndex`	the index of an array index operation is negative in a JML expression
`UndefinedNegativeSize`	the size of an array is negative in a JML expression
`UndefinedNullDeReference`	an expression being dereferenced is null in a JML expression
`UndefinedNullUnbox`	a null reference is being unboxed to a primitive in a JML expression
`UndefinedNullValue`	
`UndefinedPrecondition`	the precondition of a (pure) method being called in a JML expression does not hold
`UndefinedToolargeIndex`	the index of an array index operation is larger or equal to the array length in a JML expression
`Unreachable`	
`Writable-if`	

Table C.2: RAC warnings about assumptions (RAC only)

Warning class	Description
`ArrayInit`	
`ArgumentValue`	
`Assignment`	
`Assume`	reported when an explicit assume statement is found to be invalid
`BlockEquation`	
`BranchCondition`	
`BranchElse`	
`BranchThen`	
`Case`	
`CatchCondition`	
`DSA`	
`Havoc`	
`ImplicitAssume`	reported when an implicit assumption, generated internally by OpenJML, is found to be invalid
`LoopInvariantAssumption`	
`Lbl`	
`MethodAxiom`	
`MethodDefinition`	
`Precondition`	reported when the composite precondition of a method called within the body of the method being checked is found to be invalid during execution
`ReceiverValue`	
`Return`	
`SwitchValue`	
`Synthetic`	
`Termination`	

TODOs

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

Bibliography

Index