The OpenJML User Guide

DRAFT IN PROGRESS

David R. Cok
GrammaTech, Inc.

October 20, 2012

The most recent version of this document is available at
`http://jmlspecs.sourceforge.net/OpenJMLUserGuide.pdf`.

Contents

Part I

OpenJML

Chapter 1

Introduction

1.1 JML

This section will be added later.

1.2 OpenJDK

This section will be added later.

1.3 OpenJML

This section will be added later.

1.4 License

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

The OpenJML plug-in is a pure Eclipse plug-in, and therefore is not required to be licensed under the EPL. It does, however, call the command-line tool (in a Java sort of way), so it may be considered to be GPL v.2 as well.

In any case, the source code for both tools is available as a sourceforge project at `http://jmlspecs.svn.sourceforge.net/viewvc/jmlspecs/OpenJML/`.

Chapter 2

The command-line tool

2.1 Installation and System Requirements

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

- openjml.jar - the main jar file for the application
- jmlruntime.jar - a library needed on the classpath when running OpenJML on Java files
- jmlspecs.jar - a library containing specification files
- openjml-template.properties - a sample file, which should be copied and renamed `openjml.properties`, containing definitions of properties whose values depend on your local system
- LICENSE.rtf - a copy of the modified GPL license that applies to OpenJDK and OpenJML
- epl-v10.html - a copy of the EPL license
- OpenJMLUserGuide.pdf - this document

You can run OpenJML in a Java 1.7 JRE or, with a bit of work-around, in a Java 1.6 JRE. [1]

Java 1.7 Java 1.7 is not quite released, but you can obtain a version suitable for running OpenJML from these locations:

- for Windows and Linux: `http://dlc.sun.com.edgesuite.net/jdk7/binaries/index.html`. For testing on Windows/Cygwin, I have been using build 103, from July 2010, which you can download from `http://jmlspecs.sourceforge.net/openjdk-7-ea-src-b103-29_jul_2010.zip`
- for MacOS X: `http://formalmethods.insttech.washington.edu/software/openjml.html`

Note that the 1.7 JRE must be the current JRE in the system or the shell in which you run OpenJML.

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

Java 1.6 If you use Java 1.6, you need to add the `openjml.jar` library at the beginning of the bootclasspath, as shown in the next section.

[1] This situation appears to result from how the Java class loader handles static fields, such as Enum constants.

2.2 Running OpenJML

To run OpenJML using a Java 1.7 VM, use this command line. Here $OpenJML designates the folder in which the `openjml.jar` file resides.

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

Here <*files*> and <*options*> stand for text described below.

The following command is currently a viable alternative as well.

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

The valid options are listed in Table 2.1 and are described in subsections below.

For a 1.6 VM (on Windows/Cygwin only), use this command-line:

```
java -Xbootclasspath/p:$OPENJML/openjmlboot.jar -jar $OPENJML/openjml.jar
<options> <files>
```

2.2.1 Files

In the command templates above, <*files*> refers to a list of `.java` files. Each one 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).

You can also specify directories on the command line using the `-dir` and `-dirs` options. The `-dir` <*directory*> option indicates that the <*directory*> value (an absolute or relative path to a folder) should be understood as a folder; all `.java` or specification files within the folder are including 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 directory (if it is a directory) whose contents are processed as if listed on the command-line. Note that the `-dirs` option must be the last option.

TBD: specification files - are they processed as well?

2.2.2 Specification files

TBD : to be written

2.2.3 Java properties and the `openjml.properties` file

OpenJML uses a number of properties that may be defined in the environment; these properties are typically characteristics of the local environment and are not common across different users or different installations. An example is the the file system location of a particular solver.

The tool looks for a file named `openjml.properties` in several locations. It loads the properties it finds in each of these, in order, so later definitions will supplant earlier ones.

- System properties, including those defined with `-D` options on the command-line
- On the system classpath
- In the users home directory (the value of the Java property `user.home`
- In the current working directory (the value of the Java property `user.dir`

The properties that are currently recognized are these:

- `openjml.defaultProver` - the value is the name of the prover 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

Options specific to JML	
–	no more options
-check	typecheck only (`-command check`)
-command <*action*>	which action to do: check esc rac compile
-compile	
-counterexample	show a counterexample for failed static checks
-crossRefAssociatedInfo	
-dir <*dir*>	argument is a folder of files
-dirs	remaining arguments are folders or files
-esc	do static chacking (`-command esc`)
-java	use the native OpenJDK tool
-jmldebug	very verbose output (includes -progress)
-jmlverbose	JML-specific verbose output
-keys	
-method	
-noCheckSpecsPath	ignore non-existent specs path entries
-noPurityCheck	do not check for purity
-noInternalSpecs	do not add internal specs library to specspath
-noInternalRuntime	do not add internal runtime library to classpath
-noJML	ignore JML constructs
-nonnullByDefault	values are not null by default
-nullableByDefault	values may be null by default
-progress	
-rac	compile runtime assertion checks (`-command rac`)
-roots	
-showNotImplemented	warn if feature not implemented
-specspath	location of specs files
-stopIfParseErrors	stop if there are any parse errors
-subexpressions	show subexpression detail for failed static checks
-trace	show a trace for failed static checks

Options inherited from Java	
-Akey	
-bootclasspath <*path*>	See Java documentation.
-classpath <*path*>	location of class files
-cp <*path*>	location of class files
-d <*directory*>	location of output class files
-encoding <*encoding*>	
-endorsedirs <*dirs*>	
-extdirs <*dirs*>	
-deprecation	
-g	
-help	output 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 2.1: OpenJML options. See the text for more detail on each option.

The distribution includes a file named `openjml-template.properties` that contains stubs for all the recognized options. You should copy that file, rename it as `openjml.properties`, and edit it to reflect your system configuration. (Do not commit your system's `openjml.properties` file into the OpenJML shared SVN repository.)

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

A common source of confusion is the various different paths used to find files, specs and classes in OpenJML. OpenJML is a Java application and thus a classpath is used to find the classes that constitute the 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 not of much concern to OpenJML, but is the classpath that Java users will be familiar with. The value is implicitly given in the `-jar` form of the command. The application classpath is explicitly given in the alternate form of the command, and it may be omitted; if it is omitted, the value of the system property `CLASSPATH` is used and it must contain the `openjml.jar` library.

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

  `-classpath` <*path*>

  or

  `-cp` <*path*>

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

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

  or

  ```
  java -cp openjml.jar org.jmlspecs.openjml.Main -cp <openjml-classpath> ...
  ```

  If the OpenJML classpath is not specified, its value is obtained from 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 files. It is specified with the `-spacspath` *<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 specifications to the specspath can be disabled by using the `-noInternalSpecs` option. For example. if you have your own set of specification files that you want to use instead of the internal library, then you should use the `-noInternalSpecs` option and a `-specspath` option with a path that includes your own specification library.

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

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

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

The internal specifications are written for programs that conform to Java 1.7. [TBD - change this to adhere to the `-source` option?] [TBD - what about the specs in jmlspecs for different source levels.]

2.2.5 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.] [TBD: Actually, as of this date, the old mechanism is still in place and the new one still in progress.]

2.2.6 Annotations and the runtime library

JML uses Java annotations as introduced in Java 1.6. Those annotation classes are in the package `org.jmlspecs.annotation`. In order for files using these annotations to be processed by Java, the annotation classes must be on the classpath. They may also be required when a compiled Java program that uses such annotations is executed. In addition, running a program that has JML runtime assertion checks compiled in will require the presence of runtime classes that define utility functions used by the assertion checking code.

Both the annotation classes and the runtime checking classes are provided in a library named `jmlruntime.jar`. The distribution of OpenJML contains this library, as well as containing a version of the library within `openjml.jar`. The OpenJML is applied to a set of classes, by default it finds a version of the runtime classes and appends the location of the runtime classes to the classpath.

You can prevent OpenJML from automatically adding `jmlruntime.jar` to the classpath with the option `-noInternalRuntime`. If you use this option, then you will have to supply your own annotation classes and (if using Runtime Assertion Checking) runtime utility classes on the classpath. You may wish to do this, for example, if you have newer versions of the annotation classes that you are experimenting with. You could simply put them on the classpath, since they would be in front of the automatically added classes and used in favor of default versions; however, if you want to be sure that the default version are not present, use the `-noInternalRuntime` option.

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

2.2.7 Options: Information and debugging

- -help : prints out help information about the command-line options
- -version : prints out the version of the OpenJML tool
- -verbose : prints out verbose information about the Java processing
- -jmlverbose : prints out verbose information about the JML processing (includes -verbose)
- -progress :
- -jmldebug : prints out (voluminous) debugging information

2.2.8 Options: JML tools

- -command <*tool*> : initiates the given function; the value of <*tool*> may be one of `check`, `ese`, `rac`, TBD. The default is to use the OpenJML tool to do only typechecking of Java and JML in the source files.
- -check : causes OpenJML to do only type-checking of the Java and JML in the input files
- -compile
- -esc : causes OpenJML to do (type-checking and) static checking of the JML specifications against the implementations in the input files
- -rac
- -java : causes OpenJML to ignore all OpenJML extensions and use only the core OpenJDK functionality, so the tool should run precisely like the OpenJDK javac tool
- -noJML : causes OpenJML to use its extensions but to ignore all JML constructs (TBD - does this still recognize -check, -compile?)

TBD: jmldoc?

2.2.9 Options relating to Java version

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

2.2.10 Options: Java compiler options controlling output

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

2.2.11 Options related to Static Checking

- -counterexample
- -trace
- -subexpressions
- -method

2.2.12 Options related to parsing and typechecking

- -Werror
- -nowarn
- -stopIfParseError
- -noCheckSpecsPath
- -noPurityCheck
- -nonnullbydefault
- -nullablebydefault
- -keys

2.2.13 Options related to annotation processing

- -proc
- -processor
- -processorpath

2.2.14 Other JML Options

- -showNotImplemented
- -crossRefAssociatedInfo
- -roots

2.2.15 Other Java Options

These options are unchanged from their meaning and use in the javac tool:

- -Akey
- -J
- -X
- -implicit
- -bootclasspath
- -deprecation
- -encoding
- -endorsedirs
- -extdirs
- -g

This section will be completed later.

Chapter 3

The Eclipse Plug-in

Since OpenJML operates on Java files, it is natural that it be integrated into the Eclipse IDE. There is a conventional Eclipse plug-in that encapsulates the OpenJML command-line tool and integrates it with the Eclipse Java development environment.

3.1 Installation and System Requirements

Your system must have the following:

- A Java 1.7 JRE as described in section 2.2 (there is no 1.6 work-around for the plug-in). 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 1.7 JRE is set by the system at startup.
- Eclipse 3.6 or later

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

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

If the plug-in is successfully installed, a yellow coffee cup (the JML icon) will appear in the menubar (along with other menubar items). The installation will fail (without obvious error messages), if the underlying Java VM is not a suitable Java 1.7 VM.

3.2 GUI Features

This section will be added later.

Chapter 4

OpenJML tools

4.1 Parsing and Type-checking

This section will be added later.

4.2 Static Checking and Verification

This section will be added later.

4.3 Runtime Assertion Checking

This section will be added later.

4.4 Generating Documentation

This section will be added later.

4.5 Generating Specification File Skeletons

This section will be added later.

4.6 Generating Test Cases

This section will be added later.

Chapter 5

Using OpenJML and OpenJDK within user programs

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

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

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

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

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

Example source code The subsections that follow contain many source code examples. Small source code snippets are shown inin-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:

```
// DemoHelloWorld.java
public class DemoHelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

```
Hello, World!
```

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 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 compiling and running 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 on on the PATH java DemoHelloWorld
```

5.1 Concepts

5.1.1 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`. 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 constitute the source code created in that context.

There is little need for the user to create or manipulate Contexts. However it is essential 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]

[1]The OpenJDK software was designed as a command-line tool, in which all memory is reclaimed when the process exits.

5.1.2 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 progrmamatically, 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 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.]

5.1.3 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 5.1 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

5.1.4 Object Factories

5.1.5 Abstract Syntax Trees

5.1.6 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 factory) for that context. The scanner and parser are treated slightly differently: there is a singleton instance of a scanner factory and a parser

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.

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, DiagnosticPositionSES
Diagnostic<T>	JCDiagnostic	
	com.sun.tools.javac.main.JavaCompiler	JmlCompiler

Table 5.1: Interfaces and Classes

factory, but a new instance of the scanner and the parser are created for each compilation unit compiled. Tables 5.2 and 5.3 list the tools most likely to be encounterded when programming with OpenJML.

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

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

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

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

TBD: Is JmlTreeInfo still used

5.2 OpenJML operations

5.2.1 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
```

Each argument that would appear on the command-line is a separate argument to `execute`. All informa-

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 5.2: 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 hline creating diagnostic message objects	Check, JmlCheck JCDiagnostic.Factory	

Table 5.3: Some of the other registered tools

tional 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 is shown below:

```
// DemoExecute.java
import org.jmlspecs.openjml.*;

public class DemoExecute {

    public static void main(String[] argv) {
        try {
            IAPI m = Factory.makeAPI();
            String[] args = new String[]{"-check","src/demo/Err.java"};
            int retcode = m.execute(args);
            System.out.println("Return Code: " + retcode);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

```
DemoExecute.java:9: error: ';' expected
            String[] args = new String[]{"-check","src/demo/Err.java"}
                                                                      ^
1 error
```

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:

```
// DemoExecute2.java
import org.jmlspecs.openjml.*;
import javax.tools.*;

class MyDiagListener implements DiagnosticListener<JavaFileObject> {
    public int count = 0;
    public void report(Diagnostic<? extends JavaFileObject> diag) {
        System.out.println("Line: " + diag.getLineNumber());
        count++;
    }

}

public class DemoExecute2 {

    public static void main(String[] argv) {
        try {
            IAPI m = Factory.makeAPI();
            MyDiagListener listener = new MyDiagListener();
            int retcode = m.execute(new java.io.PrintWriter(System.out), listener,
                    "-check","-noPurityCheck","src/demo/Err.java");
            System.out.println("Errors: " + listener.count);
            System.out.println("Return code: " + retcode);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

```
Line: 6
Line: 4
Errors: 2
Return code: 1
```

5.2.2 Parsing

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

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

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

Parsing individual files

There are two methods for parsing an individual file. The basic method takes a `JavaFileObject` as input and produces an AST. The convenience method takes a filename as input and produces an AST. The methods of section 5.1.1 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.

```

// DemoParseSingle.java
import javax.tools.JavaFileObject;

import org.jmlspecs.openjml.*;

public class DemoParseSingle {

    public static void main(String[] argv) {
        try {
            IAPI m = Factory.makeAPI();
            String f1 = "src/demo/A.java";
            JavaFileObject f2 = m.makeJFOfromFilename("specs/demo/A.jml");
            JmlTree.JmlCompilationUnit ast1 = m.parseSingleFile(f1);
            JmlTree.JmlCompilationUnit ast2 = m.parseSingleFile(f2);
            m.attachSpecs(ast1,ast2);
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}

```

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 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) not 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.

5.2.3 Type-checking

5.2.4 Static checking

5.2.5 Compiling run-time checks

5.2.6 Creating JML-enhanced documentation

5.3 Working with ASTs

5.3.1 Printing parse trees

TBD

5.3.2 Source location information

TBD

5.3.3 Exploring parse trees with Visitors

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

There are three modes of traversing an AST.

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

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

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

[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.

In the example that follows, the scan method of the Visitor is modified to 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.

```
// DemoWalkTree1.java
import org.jmlspecs.openjml.*;

import com.sun.tools.javac.tree.JCTree;

public class DemoWalkTree1 {

    static class Walker extends JmlTreeScanner {

        int nodes = 0;
        int jmlopcount = 0;
        int allopcount = 0;

        @Override
        public void scan(JCTree node) {
            if (node != null) System.out.println("Node: " + node.getClass());
            if (node != null) nodes++;
            super.scan(node);
        }

        @Override
        public void visitJmlBinary(JmlTree.JmlBinary that) {
            jmlopcount++;
            allopcount++;
            super.visitJmlBinary(that);
        }

        @Override
        public void visitBinary(JCTree.JCBinary that) {
            allopcount++;
            super.visitBinary(that);
        }

    }
    public static void main(String[] argv) {
        try {
            IAPI m = Factory.makeAPI();
            Walker visitor = new Walker();
            JCTree.JCExpression expr = m.parseExpression("(a+b)*c", false);
            visitor.scan(expr);
            System.out.println("Counts: " + visitor.nodes + " " +
                visitor.allopcount + " " + visitor.jmlopcount);
            expr = m.parseExpression("a <==> \\result", true);
            visitor.scan(expr);
            System.out.println("Counts: " + visitor.nodes + " " +
                visitor.allopcount + " " + visitor.jmlopcount);

        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

```
Node: class com.sun.tools.javac.tree.JCTree$JCBinary
Node: class com.sun.tools.javac.tree.JCTree$JCParens
Node: class com.sun.tools.javac.tree.JCTree$JCBinary
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Counts: 6 2 0
Node: class org.jmlspecs.openjml.JmlTree$JmlBinary
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlSingleton
Counts: 9 3 1
```

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.

```
// DemoWalkTree2.java
import org.jmlspecs.openjml.*;

import com.sun.tools.javac.tree.JCTree;

public class DemoWalkTree2 {

    static class Walker extends JmlTreeScanner {

        public Walker(int mode) {
            super(mode);
        }

        int nodes = 0;
        int jmlopcount = 0;
        int allopcount = 0;

        @Override
        public void scan(JCTree node) {
            if (node != null) System.out.println("Node: " + node.getClass());
            if (node != null) nodes++;
            super.scan(node);
        }

        @Override
        public void visitJmlBinary(JmlTree.JmlBinary that) {
            jmlopcount++;
            allopcount++;
            super.visitJmlBinary(that);
        }

        @Override
        public void visitBinary(JCTree.JCBinary that) {
            allopcount++;
            super.visitBinary(that);
        }

    }
    public static void main(String[] argv) {
        try {
            java.io.File f = new java.io.File("src/demo/A.java");
            IAPI m = Factory.makeAPI("-specspath","specs","-sourcepath","src","-noPurityCheck");
            JmlTree.JmlCompilationUnit expr = m.parseFiles(f).get(0);
            Walker visitor = new Walker(Walker.AST_JAVA_MODE);
            visitor.scan(expr);
            System.out.println("Counts: " + visitor.nodes + " " +
                visitor.allopcount + " " + visitor.jmlopcount);
            visitor = new Walker(Walker.AST_JML_MODE);
            visitor.scan(expr);
            System.out.println("Counts: " + visitor.nodes + " " +
                visitor.allopcount + " " + visitor.jmlopcount);
            try {
                visitor = new Walker(Walker.AST_SPEC_MODE);
                visitor.scan(expr);
                System.out.println("Counts: " + visitor.nodes + " " +
                    visitor.allopcount + " " + visitor.jmlopcount);
```

```
Node: class org.jmlspecs.openjml.JmlTree$JmlCompilationUnit
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlClassDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class com.sun.tools.javac.tree.JCTree$JCPrimitiveTypeTree
Node: class com.sun.tools.javac.tree.JCTree$JCBlock
Counts: 8 0 0
Node: class org.jmlspecs.openjml.JmlTree$JmlCompilationUnit
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlClassDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodDecl
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodSpecs
Node: class org.jmlspecs.openjml.JmlTree$JmlSpecificationCase
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodClauseExpr
Node: class com.sun.tools.javac.tree.JCTree$JCBinary
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class com.sun.tools.javac.tree.JCTree$JCLiteral
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class com.sun.tools.javac.tree.JCTree$JCPrimitiveTypeTree
Node: class com.sun.tools.javac.tree.JCTree$JCBlock
Node: class org.jmlspecs.openjml.JmlTree$JmlTypeClauseDecl
Node: class org.jmlspecs.openjml.JmlTree$JmlVariableDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class com.sun.tools.javac.tree.JCTree$JCAnnotation
Node: class com.sun.tools.javac.tree.JCTree$JCFieldAccess
Node: class com.sun.tools.javac.tree.JCTree$JCFieldAccess
Node: class com.sun.tools.javac.tree.JCTree$JCFieldAccess
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class com.sun.tools.javac.tree.JCTree$JCPrimitiveTypeTree
Node: class com.sun.tools.javac.tree.JCTree$JCLiteral
Counts: 25 1 0
Node: class org.jmlspecs.openjml.JmlTree$JmlCompilationUnit
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlClassDecl
EXCEPTION: java.lang.RuntimeException: AST_SPEC_MODE requires that the Class be type-checked; clas
Node: class org.jmlspecs.openjml.JmlTree$JmlCompilationUnit
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlClassDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodSpecs
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class com.sun.tools.javac.tree.JCTree$JCBlock
Node: class com.sun.tools.javac.tree.JCTree$JCExpressionStatement
Node: class com.sun.tools.javac.tree.JCTree$JCMethodInvocation
Node: class com.sun.tools.javac.tree.JCTree$JCIdent
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodDecl
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodSpecs
Node: class org.jmlspecs.openjml.JmlTree$JmlSpecificationCase
Node: class com.sun.tools.javac.tree.JCTree$JCModifiers
Node: class org.jmlspecs.openjml.JmlTree$JmlMethodClauseExpr
Node: class com.sun.tools.javac.tree.JCTree$JCLiteral
```

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.

5.3.4 Creating parse trees

5.4 Working with JML specifications

5.5 Utilities

– version – context – symbols

5.6 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.

5.6.1 Adding new command-line options

5.6.2 Altering IAPI

5.6.3 Changing the Scanner

5.6.4 Enhancing the parser

5.6.5 Adding new modifiers and annotations

5.6.6 Adding new AST nodes

5.6.7 Modifying a compiler phase

Part II

JML

The definition of the Java Modeling Language is given in the JML Reference Manual[**?**]. This document does not repeat that definition in detail. However, it is

Chapter 6

Summary of JML Features

The definition of the Java Modeling Language is contained in the reference manual.[**?**] That definition will not be repeated here. However, the following sections contain comments about JML as they relate to the implementation within OpenJML.

6.1 JML Syntax

6.1.1 Syntax of JML specifications

JML specifications are contained in specially formatted Java comments: a JML specification includes everything between either (a) an opening `/*@` and closing `*/` or (b) an opening `//@` and the next line ending character (`\n` or `\r`) that is not within a string or character literal.

Such comments that occur within the body of a class or interface definition are considered to be a specification of the class, a field, or a method, depending on the kind of specification clause it is. JML specifications may also occur in the body of a method.

Obsolete syntax. In previous versions of JML, JML specifications could be placed within javadoc comments. Such specifications are no longer standard JML and are not supported by OpenJML.

6.1.2 Conditional JML specifications

JML has a mechanism for conditional specifications, based on a system of keys. A key is a Java identifier (alphanumerics, including the underscore character, and beginning with a non-digit). A conditional JML comment is guarded by one or more positive or negative keys (or both). The keys are placed just before the `@` character that is part of the opening sequence of the JML comment (the `//@` or the `/*@`). Each key is preceded by a '+' or a '-' sign, to indicate whether it is a positive or negative key, respectively. *No white-space is allowed.* If there is white-space anywhere between the initial `//` or `/*` and the first `@` character, the comment will appear to be a normal Java comment and will be silently ignored.

The keys are interpreted as follows. Each tool that processes the Java+JML input will have a means (e.g. by command-line options) to specify the set of keys that are enabled.

- If the JML annotation has no keys, the annotation is always processed.
- If there are only positive keys, the annotation is processed only if at least one of the keys is enabled.
- If there are only negative keys, the annotation is processed unless one of the keys is enabled.
- If there are both positive and negative keys, the annotation is processed only if (a) at least one of the positive keys is enabled AND (b) none of the negative keys are enabled.

JML previously defined one conditional annotation: those that began with `/*+@` or `//+@`. ESC/Java2 also defined `/*-@` and `//-@`. Both of these are now deprecated. OpenJML does have an option to enable the +-style comments.

The particular keys do not have any defined meaning. OpenJML implicitly enables the ESC key when it is performing ESC static checking; it implicitly enables the RAC key when it is performing Runtime-Assertion-Checking. Thus, for example, one can turn off a non-executable assert statement for RAC-processing by writing //-RAC@ assert ...

6.1.3 Finding specification files and the refine statement

JML allows specifications to be placed directly in the .java files that contain the implementation of methods and classes. Indeed, specifications such as assert statements or loop invariants are necessarily placed directly in a method body. Other specifications, such as class invariants and method pre- and post-conditions, may be placed in auxiliary files. For classes which are only present as .class files and not as .java files, the auxiliary file is a necessity.

Current JML allows one such auxiliary file. It is similar to the corresponding .java file except that

- it has a .jml suffix
- it contains no method bodies (method declarations are terminated with semi-colons, as if they were abstract)

The .jml file is in the same package as the corresponding .java file and has the same name, except for the suffix. If there is no source file, then there is a .jml file for each class that has a specification. [TBD - what about non public classes]

This section will be added later.

Obsolete syntax. The `refine` and `refines` statements are no longer recognized. The previous (complicated) method of finding specification files and merging the specifications from multiple files is also no longer implemented. The only specification file suffix allowed is `.jml`; the others — `.spec`, `.refines-java`, `.refines-spec`, `.refines-jml` — are no longer implemented.

In addition, the `.jml` file is sought before seeking the `.java` file; if a `.jml` file is found anywhere in the specs path, then any specifications in the `.java` file are ignored. This is a different search algorithm than was previously used.

6.1.4 JML specifications and Java annotations

This section will be added later.

6.1.5 Model import statements

This section will be added later.

6.1.6 Modifiers

This section will be added later.

- note elimination of weakly

6.1.7 Method specification clauses

This section will be added later.

6.1.8 Class specification clauses

This section will be added later.

6.1.9 Visibility of specifications

This section will be added later.

6.1.10 Statement specifications

This section will be added later.

6.1.11 Refining statement specifications

This section will be added later.

6.1.12 JML expressions

This section will be added later.

6.1.13 JML operators

This section will be added later.

6.1.14 JML types

This section will be added later.

6.1.15 JML informal comments

This section will be added later.

6.1.16 Non-Null and Nullable

This section will be added later.

6.1.17 Race condition detection

This section will be added later.

6.1.18 Arithmetic modes

This section will be added later.

6.1.19 Universe types

This section will be added later.

6.1.20 Dynamic frames

This section will be added later.

6.1.21 Code contracts

This section will be added later.

6.1.22 redundantly suffixes

This section will be added later.

6.1.23 nowarn lexical construct

This section will be added later.

6.2 Interaction with Java features

This section will be added later.

6.3 Other issues

6.3.1 Interaction with JSR-308

This section will be added later.

This section will be added later.

6.3.2 Interaction with FindBugs

This section will be added later.

Part III

Semantics and translation of Java and JML in OpenJML

Chapter 7

Introduction

- Sorted First-order-logic
  - individual subexpressions; optional expression form; optimization; usefulness for tracing
  - RAC vs. ESC
  - nomenclature

Chapter 8

Statement translations

TODO: Need to insert both RAC and ESC in all of the following.

8.1 While loop

Java and JML statement:

```
//@ invariant invariant_condition ;
//@ decreases counter ;
while (condition) {
    body
}
```

Translation: *TOOD: Needs variant condition, havoc information*

```
{
    //@ assert jmltranslate(invariant_condition) ;
    //@ assert jmltranslate(variant_condition) > 0 ;
    while (true) {
        stats(tmp,condition)
        if (!tmp) {
            //@ assume !tmp;
            break;
        }
        //@ assume tmp;
        stats(body)
    } }
```

Chapter 9

Java expression translations

9.1 Implicit or explicit arithmetic conversions

TODO

9.2 Arithmetic expressions

TODO: need arithmetic range assertions

In these, *T* is the type of the result of the operation. The two operands in binary operations are already assumed to have been converted to a common type according to Java's rules.

```
stats(tmp, - a ) ==>
    stats(tmpa, a )
    T tmp = - tmpa ;

stats(tmp, a + b ) ==>
    stats(tmpa, a )
    stats(tmpb, b )
    T tmp = tmpa + tmpb ;

stats(tmp, a - b ) ==>
    stats(tmpa, a )
    stats(tmpb, b )
    T tmp = tmpa - tmpb ;

stats(tmp, a * b ) ==>
    stats(tmpa, a )
    stats(tmpb, b )
    T tmp = tmpa * tmpb ;

stats(tmp, a / b ) ==>
    stats(tmpa, a )
    stats(tmpb, b )
    //@ assert tmpb != 0; // No division by zero
    T tmp = tmpa / tmpb ;

stats(tmp, a % b ) ==>
    stats(tmpa, a )
    stats(tmpb, b )
    //@ assert tmpb != 0; // No division by zero
    T tmp = tmpa % tmpb ;
```

9.3 Bit-shift expressions

TODO

9.4 Relational expressions

No assertions are generated for the relational operations `<` `>` `<=` `>=` `==` `!=`. The operands are presumed to have been converted to a common type according to Java's rules.

```
stats(tmp, a op b ) ==>
    stats(tmpa, a )
    stats(tmpb, b )
    T tmp = tmpa op tmpb ;
```

9.5 Logical expressions

```
stats(tmp, ! a ) ==>
    stats(tmpa, a )
    T tmp = ! tmpa ;
```

The `&&` and `||` operations are short-circuit operations in which the second operand is conditionally evaluated. Here `&` and `|` are the (FOL) boolean non-short-circuit conjunction and disjunction.

```
stats(tmp, a && b ) ==>
    boolean tmp ;
    stats(tmpa, a )
    if ( tmpa ) {
        //@ assume tmpa ;
        stats(tmpb, b )
        tmp = tmpa & tmpb ;
    } else {
        //@ assume ! tmpa ;
        tmp = tmpa ;
    }
```

```
stats(tmp, a || b ) ==>
    boolean tmp ;
    stats(tmpa, a )
    if ( ! tmpa ) {
        //@ assume ! tmpa ;
        stats(tmpb, b )
        tmp = tmpa | tmpb ;
    } else {
        //@ assume tmpa ;
    tmp = tmpa ;
}
```

An index will be added later.