Preliminary Design of JML:

A Behavioral Interface Specification Language for Java

by Gary T. Leavens, Albert L. Baker, and Clyde Ruby

TR #98-06-rev29
June 1998, revised July, November 1998,
January, April, June, July, August, December 1999,
February, May, July, December 2000,
February, April, May, August 2001,
June, August, October, December 2002,
April, May, September, November 2003,
June, November, December 2004,
February, April, July 2005,
January 2006

Keywords: Behavioral interface specification, Java, JML, Eiffel, Larch, model-based specification, assertion, precondition, postcondition, frame.

2000 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, tools, theory, Larch, Eiffel, JML, ESC/Java; D.2.4 [*Software Engineering*] Software/Program Verification — assertion checkers, class invariants, formal methods, programming by contract; D.2.7 [*Software Engineering*] Distribution and Maintenance — documentation; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — assertions, invariants, logics of programs, pre- and post-conditions, specification techniques.

Appears as *ACM SIGSOFT Software Engineering Notes*, 31(3):1-38, March 2006.

Department of Computer Science, Iowa State University
226 Atanasoff Hall
Ames, Iowa 50011-1041, USA

Table of Contents

1 Introduction

Abstract

JML is a behavioral interface specification language tailored to Java(TM). Besides pre- and postconditions, it also allows assertions to be intermixed with Java code; these aid verification and debugging. JML is designed to be used by working software engineers; to do this it follows Eiffel in using Java expressions in assertions. JML combines this idea from Eiffel with the model-based approach to specifications, typified by VDM and Larch, which results in greater expressiveness. Other expressiveness advantages over Eiffel include quantifiers, specification-only variables, and frame conditions.

This paper discusses the goals of JML, the overall approach, and describes the basic features of the language through examples. It is intended for readers who have some familiarity with both Java and behavioral specification using pre- and postconditions.

JML stands for "Java Modeling Language" [Leavens-Baker-Ruby99]. JML is a *behavioral interface specification language* (BISL) [Wing87] designed to specify Java [Arnold-Gosling-Holmes00] [Gosling-etal00] modules. Java *modules* are classes and interfaces.

The main goal of our research on JML is to better understand how to make BISLs (and BISL tools) that are practical and effective for production software environments. In order to understand this goal, and the more detailed discussion of our goals for JML, it helps to define more precisely what a behavioral interface specification is. After doing this, we return to describing the goals of JML, and then give a brief overview of the tool support for JML and an outline of the rest of the paper.

1.1 Behavioral Interface Specification

As a BISL, JML describes two important aspects of a Java module:

- its *interface*, which consists of the names and static information found in Java declarations, and
- its *behavior*, which tells how the module acts when used.

BISLs are inherently language-specific [Wing87], because they describe interface details for clients written in a specific programming language, For example, a BISL tailored to C++, such as Larch/C++ [Leavens97c], describes how to use a module in a C++ program. A Larch/C++ specification cannot be implemented correctly in Java, and a JML specification cannot be correctly implemented in C++ (except for methods that are specified as native code).

JML specifications can either be written in separate files or as annotations in Java program files. To a Java compiler such annotations are comments that are ignored [Luckham-vonHenke85] [Luckham-etal87] [Rosenblum95] [Tan94] [Tan95]. This allows JML specifications, such as the specification below, to be embedded in Java program files. Consider the following simple example of a behavioral interface specification in JML, written as annotations in a Java program file, '`IntMathOps.java`'.

```
public class IntMathOps {                                      // 1
                                                               // 2
  /*@ public normal_behavior                                   // 3
    @   requires y >= 0;                                       // 4
    @   assignable \nothing;                                   // 5
    @   ensures 0 <= \result                                   // 6
    @        && \result * \result <= y                         // 7
    @        && ((0 <= (\result + 1) * (\result + 1))          // 8
    @            ==> y < (\result + 1) * (\result + 1)); // 9
    @*/                                                        //10
  public static int isqrt(int y)                               //11
  {                                                            //12
    return (int) Math.sqrt(y);                                 //13
  }                                                            //14
}                                                              //15
```

The specification above describes a Java class, `IntMathOps` that contains one static method (function member) named `isqrt`. The single-line comments to the far right (which start with `//`) give the line numbers in this specification; they are ignored by both Java and JML. Comments with an immediately following at-sign, `//@`, or, as on lines 3–10, C-style comments starting with `/*@`, are *annotations*. Annotations are treated as comments by a Java compiler, but JML processes the text of an annotation. The text of an annotation is either the remainder of a line following `//@` or the characters between the annotation markers `/*@` and `@*/`. In the second form, at-signs (`@`) at the beginning of lines are ignored; they can be used to help the reader see the extent of an annotation.

In the above specification, interface information is declared in lines 1 and 11. Line 1 declares a class named `IntMathOps`, and line 11 declares a method named `isqrt`. Note that all of Java's declaration syntax is allowed in JML, including, on lines 1 and 11, that the names declared are `public`, that the method is `static` (line 11), that its return type is `int` (line 11), and that it takes one `int` argument.

Such interface declarations must be found in a Java module that correctly implements this specification. This is automatically the case in the file '`IntMathOps.java`' shown above, since that file also contains the implementation. In fact, when Java annotations are embedded in '`.java`' files, the interface specification is the actual Java source code.

To be correct, an implementation must have both the specified interface and the specified behavior. In the above specification, the behavioral information is specified in the annotation text on lines 3–10.[1] The keywords `public normal_behavior` are used to say that the specification is intended for callers (hence "public"), and that when the precondition is satisfied a call must return normally, without throwing an exception (hence "normal"). In such a public specification, only names with public visibility may be used.[2] On line

[1] In JML method specifications must be placed either before the method's header, as shown above, or between the method's header and its body. In this document, we always place the specification before the method header. This convention is followed by many Java tools, in particular by Javadoc; It has the advantage of working in all cases, even when the method has no body.

[2] In a protected specification, both public and protected identifiers can be used. In a specification with default (i.e., no) visibility specified, which corresponds to Java's default visibility, public and protected

4 is a precondition, which follows the keyword `requires`.[3] On line 5 is frame condition, which says that this method, when called, does not assign to any locations. On lines 6–9 is a postcondition, which follows the keyword `ensures`.[4] The precondition says what must be true about the arguments (and other parts of the state); if the precondition is true, then the method must terminate normally in a state that satisfies the postcondition. This is a contract between the caller of the method and the implementor [Hoare69] [Jones90] [Jonkers91] [Guttag-Horning93] [Meyer92a] [Meyer97] [Morgan94]. The caller is obligated to make the precondition true, and gets the benefit of having the postcondition then be satisfied. The implementor gets the benefit of being able to assume the precondition, and is obligated to make the postcondition true in that case.

In general, pre- and postconditions in JML are written using an extended form of Java expressions. In this case, the only extension visible is the keyword `\result`, which is used in the postcondition to denote the value returned by the method. The type of `\result` is the return type of the method; for example, the type of `\result` in `isqrt` is `int`. The postcondition says that the result is an integer approximation to the square root of `y`. The first conjunct on line 6, `0 <= \result` say that the result is non-negative. The second and third conjuncts state that the result is an integer approximation to the square root of the argument `y`. The second conjunct, on line 7, says that the result squared is no larger than the argument, `y`. The third conjunct, on lines 8–9, is an implication; it has two expressions connected by `==>`, which means implication in JML. This implication says that if the result plus one squared is non-negative, then the result plus one squared is strictly larger than `y`.[5] Note that the behavioral specification does not give an algorithm for finding the square root.

Method specifications may also be written in Java's documentation comments. The following is an example. The part that JML sees is enclosed within the HTML "tags" `<jml>` and `</jml>`.[6] As in this example, one can use surrounding tags `<pre>` and `</pre>` to tell javadoc to ignore what JML sees, and to leave the formatting of it alone. The `<pre>` and `</pre>` tags are not required by JML tools (including jmldoc, which does a better job of formatting specifications than does javadoc).

---

identifiers can be used, as well as identifiers from the same package with default visibility. A private specification can use any identifiers that are available. The privacy level of a method specification cannot allow more access than the method being specified. Thus a public method may have a private specification, but a private method may not have a public specification.

[3] The keyword `pre` can also be used as a synonym for `requires`.

[4] The keyword `post` can also be used as a synonym for `ensures`.

[5] The result plus one squared will become negative if the result is larger than 46340, due to integer overflow. Patrice Chalin pointed out that in an earlier version of this specification there were overflow problems [Chalin02]. In Java integer arithmetic, one plus the maximum integer is the minimum integer. This specification deals with such problems by limiting the result to be a positive integer and by the implication on lines 8-9. See the specification of `IntMathOps2` below for another way to deal with these problems.

[6] Since HTML tags are not case sensitive, in this one place JML is also not case sensitive. That is, the syntax also permits the tags `<JML>`, `</JML>`. For compatibility with ESC/Java, JML also supports the tags `<esc>`, `</esc>`, `<ESC>`, and `</ESC>`.

```
public class IntMathOps4 {

  /** Integer square root function.
   * @param y the number to take the root of
   * @return an integer approximating
   *         the positive square root of y
   * <pre><jml>
   *   public normal_behavior
   *     requires y >= 0;
   *     assignable \nothing;
   *     ensures 0 <= \result
   *        && \result * \result <= y
   *        && ((0 <= (\result + 1) * (\result + 1))
   *            ==> y < (\result + 1) * (\result + 1));
   * </jml></pre>
   **/
  public static int isqrt(int y)
  {
     return (int) Math.sqrt(y);
  }
}
```

Because we expect most of our users to write specifications in Java code files, most of our examples will be given as annotations in '`.java`' files as in the specifications above. However, it is possible to use JML to write documentation in separate, non-Java files, such as the file '`IntMathOps2.jml-refined`' below. Since these files are not Java program files, JML requires the user to omit the code for concrete methods in such a file (except that code for "model" methods can be present, see Section 2.3.1 [Purity], page 29). The specification below shows how this is done, using a semicolon (`;`), as in a Java abstract method declaration.

```
//@ model import org.jmlspecs.models.*;

public /*+@ spec_bigint_math @+*/ class IntMathOps2 {

 /*@ public normal_behavior
   @   requires y >= 0;
   @   assignable \nothing;
   @   ensures -y <= \result && \result <= y;
   @   ensures \result * \result <= y;
   @   ensures y < (Math.abs(\result) + 1)
   @               * (Math.abs(\result) + 1);
   @*/
  public static int isqrt(int y);
}
```

Besides files with suffixes of '`.jml-refined`' or '`.jml`', JML also works with files with the suffixes '`.spec`' and '`.spec-refined`'. All these files use Java's syntax, and one must use annotation markers just as in a '`.java`' file. However, since these kinds of files files are not Java files, in such a file one must also omit the code for concrete, non-model methods.

The specification of `IntMathOps2` below is written in `spec_bigint_math` mode [Chalin04]. This means that integer mathematics inside the specifications in the class `IntMathOps2` are done in infinite precision arithmetic, instead of the usual Java arithmetic. This leads to a simpler specification, especially in the ensures clause.[7]

The above specification also demonstrates that ensures clauses can be repeated in a specification. In `IntMathOps2`'s specification of `isqrt`, there are three ensures clauses; all of them must be satisfied. Thus the meaning is the same as the conjunction of all of the postconditions specified in the individual ensures clauses. This specification is also more underspecified than the specifications given previously, as it allows negative numbers to be returned as results.

The above specification would be implemented in the file '`IntMathOps2.java`', which is shown below. This file contains a `refine` clause, which tells the reader of the '`.java`' file what is being refined and the file in which to find its specification.

```
//@ refine "IntMathOps2.jml-refined";

//@ model import org.jmlspecs.models.*;

public class IntMathOps2 {

  public static int isqrt(int y)
  {
     return (int) Math.sqrt(y);
  }
}
```

To summarize, a behavioral interface specification describes both the interface details of a module, and its behavior. The interface details are written in the syntax of the programming language; thus JML uses the Java declaration syntax. The behavioral specification uses pre- and postconditions.

1.2 Lightweight Specifications

Although we find it best to illustrate JML's features in this paper using specifications that are detailed and complete, one can use JML to write less detailed specifications. In particular, one can use JML to write "lightweight" specifications (as in ESC/Java). The syntax of JML allows one to write specifications that consist of individual clauses, so that one can say just what is desired. More precisely, a *lightweight* specification is one that does not use a behavior keyword (like `normal_behavior`). By way of contrast, we call a specification a *heavyweight* specification if it uses one of the behavior keywords.

[7] Because the current ESC/Java2 tool does not understand `spec_bigint_math` mode, the specification uses uses annotation markers `/*+@` and `@+*/`. These markers are understood by the ISU JML tools, but are considered to be comments by ESC/Java2.

For example, one might wish to specify just that `isqrt` should be called only on positive arguments, but not want to be bothered with saying anything formal about the locations that can be assigned to by the method or about the result. This could be done as shown below. Notice that the only specification given below is a single `requires` clause. Since the specification of `isqrt` has no behavior keyword, it is a lightweight specification.

```
public class IntMathOps3 {

  //@ requires y >= 0;
  public static int isqrt(int y)
  {
    return (int) Math.sqrt(y);
  }
}
```

What is the access restriction, or privacy level, of such a lightweight specification? The syntax for lightweight specifications does not have a place to specify the privacy level, so JML assumes that such a lightweight specification has the same level of visibility as the method itself. (Thus, the specification below is implicitly `public`.) What about the omitted parts of the specification, such as the ensures clause? JML assumes nothing about these. In the example below when the precondition is met, an implementation might either signal an exception or terminate normally, so this specification technically allows exceptions to be thrown. However, the gain in brevity often outweighs the need for this level of precision.

JML has a semantics that allows most clauses to be sensibly omitted from a specification. When the `requires` clause is omitted, for example, it means that no requirements are placed on the caller. When the `assignable` clause is omitted, it means that nothing is promised about what locations may not be assigned to by the method; that is, the method may assign to all locations that it can otherwise legally assign to. When the `ensures` clause is omitted, it means that nothing is promised about the state resulting from a method call. See Appendix A [Specification Case Defaults], page 62, for the default meanings of various other clauses.

1.3 Goals

As mentioned above, the main goal of our research is to better understand how to develop BISLs (and BISL tools) that are practical and effective. We are concerned with both technical requirements and with other factors such as training and documentation, although in the rest of this paper we will only be concerned with technical requirements for the BISL itself. The practicality and effectiveness of JML will be judged by how well it can document reusable class libraries, frameworks, and Application Programming Interfaces (APIs).

We believe that to meet the overall goal of practical and effective behavioral interface specification, JML must meet the following subsidiary goals.

- JML must be able to document the interfaces and behavior of existing software, regardless of the analysis and design methods used to create it.

  If JML were limited to only handling certain Java features, certain kinds of software, or software designed according to certain analysis and design methods, then some APIs would not be amenable to documentation using JML. This would mean that some existing software could not be documented using JML. Since the effort put into writing such documentation will have a proportionally larger payoff for software that is more widely reused, it is important to be able to document existing software components.

(However, it should be noted that we make some exceptions to this goal. One is that JML requires that all subtypes be behavioral subtypes [Dhara-Leavens96] [Leavens97c] [Wing87] of their supertypes. This is done because otherwise one cannot reason modularly about programs that use subtyping and dynamic dispatch. Another is that we specify `Object`'s method `equals` as a pure method, which prohibits even benevolent side effects in any `equals` method that takes an `Object` as an argument. This is done to permit purity checking for collection classes that contain objects as members and use `equals` to compare them, as in the collection types found in `java.util`.)

- The notation used in JML should be readily understandable by Java programmers, including those with only standard mathematical training.

  A preliminary study by Finney [Finney96] indicates that graphic mathematical notations, such as those found in Z [Hayes93] [Spivey92] [Woodcock-Davies96] may make such specifications hard to read, even for programmers trained in the notation. This accords with our experience in teaching formal specification notations to programmers. Hence, our strategy for meeting this goal has been to shun most special-purpose mathematical notations in favor of Java's own expression syntax.

- The language must be capable of being given a rigorous, formal semantics, and must also be amenable to tool support.

  This goal also helps ensure that the specification language does not suffer from logical problems, which would make it less useful for static analysis, prototyping, and testing tools.

We also have in mind a long range goal of a specification compiler, that would produce prototypes from specifications that happen to be constructive [Wahls-Leavens-Baker00].

Our partners at Compaq SRC and the University of Nijmegen have other goals in mind. At Compaq SRC, the goal is to make static analysis tools for Java programs that can help detect bugs. At the University of Nijmegen, the goal is to be able to do full program verification on Java programs.

As a general strategy for achieving these goals, we have tried to blend the Eiffel [Meyer92a] [Meyer92b] [Meyer97], Larch [Wing87] [Wing90a] [Guttag-Horning93] [LeavensLarchFAQ], and refinement calculus [Back88] [Back-vonWright98] [Morgan-Vickers94] [Morgan94] approaches to specification. From Eiffel we have taken the idea that assertions can be written in a language that is based on Java expressions. We also adapt the "`old`" notation from Eiffel, which appears in JML as `\old`, instead of the Larch-style annotation of names with state functions. However, Eiffel specifications, as written by Meyer, are typically not as detailed as model-based specifications written, for example, in Larch BISLs or in VDM-SL [Fitzgerald-Larsen98] [ISO96] [Jones90]. Hence, we have combined these approaches, by using syntactic ideas from Eiffel and semantic ideas from model-based specification languages.

JML also has some other differences from Eiffel (and its cousins Sather and Sather-K). The most important is the concept of specification-only declarations. These declarations allow more abstract and exact specifications of behavior than is typically done in Eiffel; they allow one to write specifications that are similar to the spirit of VDM or Larch BISLs.

A major difference is that we have extended the syntax of Java expressions with quantifiers and other constructs that are needed for logical expressiveness, but which are not always executable. Finally, we ban side-effects and other problematic features of code in assertions.

On the other hand, our experience with Larch/C++ has taught us to adapt the model-based approach in two ways, with the aim of making it more practical and easy to learn. The first adaptation is again the use of specification-only model variables. An object will thus have (in general) several such *model fields*, which are used only for the purpose of describing, abstractly, the values of objects. This simplifies the use of JML, as compared with most Larch BISLs, since specifiers (and their readers) hardly ever need to know about algebraic-style specification. It also makes designing a model for a Java class or interface similar, in some respects, to designing an implementation data structure in Java. We hope that this similarity will make the specification language easier to understand. (This kind of model also has some technical advantages that will be described below.)

The second adaptation is hiding the details of mathematical modeling behind a facade of Java classes. In the Larch approach to behavioral interface specification [Wing87], the mathematical notation used in assertions is presented directly to the specifier. This allows the same mathematical notation to be used in many different specification languages. However, it also means that the user of such a specification language has to learn a notation for assertions that is different than their programming language's notation for expressions. In JML we use a compromise approach, hiding these details behind Java classes. These classes have objects with many "pure" methods, in the sense that they do not use side-effects (at least not in any observable way). Such classes are intended to present the underlying mathematical concepts using Java syntax. Besides insulating the user of JML from the details of the mathematical notation, this compromise approach also insulates the design of JML from the details of the mathematical logic used for theorem proving.

We have generally taken features wholesale from the refinement calculus [Back88] [Back-vonWright98] [Morgan-Vickers94] [Morgan94]. Our adaptation of it consists in blending it with the idea of interface specification and adding features for object-oriented programming. We are using the adaptation of the refinement calculus by Büchi and Weck [Buechi-Weck00], which helps in specifying callbacks. However, since the refinement calculus is mostly needed for advanced specifications, in the remainder of this paper we do not discuss the JML features related to refinement, such as model programs.

1.4 Tool Support

Our partners at Compaq SRC have built a tool, ESC/Java, that does static analysis for Java programs [Leino-etal00]. ESC/Java uses a subset of the JML specification syntax, to help detect bugs in Java code. At the University of Nijmegen the LOOP tool [Huisman01] [Jacobs-etal98] is being adapted to use JML as its input language. This tool would generate verification conditions that could be checked using a theorem prover such as PVS or Isabelle/HOL. At the Massachusetts Institute of Technology (MIT), the Daikon invariant detector project [Ernst-etal01] is using a subset of JML to record invariants detected by runs of a program. Recent work uses ESC/Java to validate the invariants that are found.

In the rest of the section we concentrate on the tool support found in the JML release from Iowa State. Iowa State's JML release has tool support for: static type checking of specifications, run-time assertion checking, generation of HTML pages, and generation of unit testing harnesses. Use a web browser on the 'JML.html' file in the Iowa State JML release to access more detailed documentation on these tools.

1.4.1 Type Checking Specifications

Details on how to run the JML checker can be found in its manual page, which is part of the JML release. Here we only indicate the most basic uses of the checker. Running the checker with filenames as arguments will perform type checking on all the specifications contained in the given files. For example, one could check the specifications in the file 'UnboundedStack.java' by executing the following command.

```
jml UnboundedStack.java
```

One can also pass several files to the checker. For example, the following shows a handy pattern to catch all of the JML files in the current directory.

```
jml *.*j* *.*spec*
```

One can also pass directories to the JML checker, for example the following will check all the specifications in the current directory.

```
jml .
```

By default, the checker does not recurse into subdirectories, but this can be changed by using the -R option. For example, the following checks specifications in the current directory and all subdirectories.

```
jml -R .
```

To allow specifications to be split into several files and to allow documentation of code without changing existing files, the checker recognizes several filename suffixes. The checker recognizes several filename suffixes. The following are considered to be "active" suffixes: '.refines-java', '.refines-spec', '.refines-jml', '.java', '.spec', and '.jml'; There are also three "passive" suffixes: '.java-refined', '.spec-refined', and '.jml-refined'. Files with passive suffixes can be used in refinements (see Section 1.1 [Behavioral Interface Specification], page 1) but should not normally be passed explicitly to the checker on its command line. Graphical user interface tools for JML should, by default, only present the active suffixes for selection. Among files in a directory with the same prefix, but with different active suffixes, the one whose suffix appears first in the list of active suffixes above should be considered primary by such a tool.

Files with different suffixes should be connected to each other using `refines` clauses. We give several examples in the remainder of this paper.

1.4.2 Generating HTML Documentation

To generate HTML documentation that can be browsed on the web, one uses the `jmldoc` tool.[8] This tool is a replacement for `javadoc` that understands JML specifications. In addition to generating web pages for JML files and for JML annotated Java files, `jmldoc` also generates the indexes and other HTML files that surround these and provide access, in the same way that `javadoc` does.

For example, here is how we use `jmldoc` to generate the HTML pages for the MultiJava project.

[8] The `jmldoc` tool is generously provided by David Cok; thanks David!.

```
rm -fr $HOME/MJ/javadocs
jmldoc -Q -private -d $HOME/MJ/javadocs \
  -link file:/cygwin/usr/local/jdk1.4/docs/api \
  -link file:/cygwin/usr/local/antlr/javadocs \
  --sourcepath $HOME/MJ  \
  org.multijava.dis org.multijava.javadoc org.multijava.mjc \
  org.multijava.mjdoc org.multijava.util org.multijava.util.backend \
  org.multijava.util.classfile org.multijava.util.compiler \
  org.multijava.util.jperf org.multijava.util.lexgen \
  org.multijava.util.msggen org.multijava.util.optgen \
  org.multijava.util.optimize org.multijava.util.testing
```

The options used in the above invocation of `jmldoc` make jmldoc be quiet (`-Q`), document all members (including private ones) of classes and interfaces (`-private`), write the HTML files relative to '`$HOME/MJ/javadocs`' (`-d`), link to existing HTML files for the JDK and for ANTLR (`-link`), and find listed packages relative to '`$HOME/MJ`' (`--sourcepath`). More details on running `jmldoc` are available from its manual page, which is part of the JML release.

1.4.3 Run Time Assertion Checking

The JML runtime assertion checking compiler is called `jmlc`. It type checks assertions (so there is no need to run `jml` separately), and then generates a class file with the executable parts of the specified assertions, invariants, preconditions, and postconditions (and other JML constructs) checked at run-time. Its basic usage is similar to a Java compiler, as shown in the following example.

```
jmlc TestUnboundedStack.java UnboundedStack.java
```

This will produce output telling what the compiler is doing, as well as class files '`TestUnboundedStack.class`' and '`UnboundedStack.class`'.

To run or test a program compiled with `jmlc`, you should use the script `jmlrac`. The `jmlrac` script runs the resulting code with a `CLASSPATH` that includes various JAR files containing code needed for run-time assertion checking. The following is an example.

```
jmlrac org.jmlspecs.samples.stacks.TestUnboundedStack
```

Using the `jmlrac` script is necessary. If you do not use `jmlrac` to run the program, you will get errors, since the code that `jmlc` compiles expects various runtime library classes to be available.

More details on invoking `jmlc` and `jmlrac` are available from their manual pages, which are available in the JML release. See also the '`README.html`' file in the JML release for more details and troubleshooting tips. Details on the implementation of `jmlc` are found in a paper by Cheon and Leavens [Cheon-Leavens02b].

1.4.4 Unit Testing with JML

The run time assertion checker is also integrated with a tool, `jmlunit` that can write out a JUnit [Beck-Gamma98] test oracle class for given Java files. For example, to generate the classes `UnboundedStack_JML_Test` and `UnboundedStack_JML_TestData` from `UnboundedStack`, one would execute the following.

```
jmlunit UnboundedStack.java
```

The file 'UnboundedStack_JML_Test.java' will then contain code for an abstract class to drive the tests. This class uses the runtime assertion checker to decide test success or failure. (Tests are only as good as the quality of the specifications; hence the specifications must be reasonably complete to permit reasonably complete testing.)

The file 'UnboundedStack_JML_TestData.java' will contain code for the superclass of UnboundedStack_JML_Test that can be used to fill in test data for such testing. You need to fill in the test data in the code for this subclass, as described in the comments. The file 'UnboundedStack_JML_TestData.java' is produced automatically the first time you run jmlunit as described above. However, subsequent runs of jmlunit never overwrite or change an '_JML_TestData.java' file such as this if it exists. Hence it is safe to edit the file to add test data (and even additional test methods if you wish).

To run the test use the script jml-junit, as in the following example.

```
jml-junit org.jmlspecs.samples.stacks.UnboundedStack_JML_TestData
```

More details on invoking these tools can be found in their manual pages which ship with the JML release. More discussion on this integration of JML and JUnit are explained in the ECOOP 2002 paper by Cheon and Leavens [Cheon-Leavens02].

JML also provides a tool, jtest, that combines both jmlc and jmlunit. The jtest tool both compiles a class with run-time assertion checks enabled using jmlc, and also generates the test oracle and test data classes, using jmlunit.

1.5 Outline

In the next sections we describe more about JML and its semantics. See Chapter 2 [Class and Interface Specifications], page 12, for examples that show how Java classes and interfaces are specified; this section also briefly describes the semantics of subtyping and refinement. See Chapter 3 [Extensions to Java Expressions], page 55, for a description of the expressions that can be used in specifications. See Chapter 4 [Conclusions], page 60, for conclusions from our preliminary design effort. See the *JML Reference Manual* [Leavens-etal-JMLRef] for details on the syntax of JML.

2 Class and Interface Specifications

In this section we give some examples of JML class specifications that illustrate the basic features of JML.

2.1 Abstract Models

A simple example of an abstract class specification is the ever-popular `UnboundedStack` type, which is presented below. It would appear in a file named '`UnboundedStack.java`'.

```
package org.jmlspecs.samples.stacks;

//@ model import org.jmlspecs.models.*;

public abstract class UnboundedStack {

  /*@ public model JMLObjectSequence theStack;
    @ public initially theStack != null
    @                  && theStack.isEmpty();
    @*/

  //@ public invariant theStack != null;

  /*@ public normal_behavior
    @   requires !theStack.isEmpty();
    @   assignable theStack;
    @   ensures theStack.equals(
    @             \old(theStack.trailer()));
    @*/
  public abstract void pop( );

  /*@ public normal_behavior
    @   assignable theStack;
    @   ensures theStack.equals(
    @             \old(theStack.insertFront(x)));
    @*/
  public abstract void push(Object x);

  /*@ public normal_behavior
    @   requires !theStack.isEmpty();
    @   assignable \nothing;
    @   ensures \result == theStack.first();
    @*/
  public /*@ pure @*/ abstract Object top( );
}
```

The above specification contains the declaration of a model field, an invariant, and some method specifications. These are described below.

2.1.1 Model Fields

In the fourth non-blank line of '`UnboundedStack.java`', a model data field, `theStack`, is declared. Since it is declared using the JML modifier `model`, such a field is not part of the Java implementation, and must appear in an annotation; however, for purposes of the specification we treat it much like any other Java field (i.e., as a variable location). That is, we imagine that each instance of the class `UnboundedStack` has such a field.

The type of the model field `theStack` is a type designed for mathematical modeling, `JMLObjectSequence`. Objects of this type are sequences of objects. This type is provided by JML in the package `org.jmlspecs.models`, which is imported in the second non-blank line of the figure. Note that this `import` declaration is not part of the Java implementation, since it is modified by the keyword `model`. Such model imports must also appear in annotation comments. In general, any declaration form in Java can have the `model` modifier, with the same meaning. That is, a model declaration is only used for specification purposes, and does not have to appear in an implementation.

At the end of the model field's declaration above is an `initially` clause. (Such clauses are adapted from RESOLVE [Ogden-etal94] and the refinement calculus [Back88] [Back-vonWright98] [Morgan-Vickers94] [Morgan94].) Model fields cannot be explicitly initialized (and thus cannot be final), because there is no storage directly associated with them. However, one can use an `initially` clause to describe an abstract initialization for a model field. Initially clauses can be attached to any field declaration, including non-model fields, and permit one to constrain the initial values of such fields. Knowing something about the initial value of the field permits data type induction [Hoare72a] [Wing83] for abstract classes and interfaces. The `initially` clause must be true of the field's starting value. That is, all reachable objects of the type `UnboundedStack` must have been created as empty stacks and subsequently modified using the type's methods.

2.1.2 Invariants

Following the model field declaration is an invariant. An invariant does not have to hold during the execution of an object's methods, but it must hold, for each reachable object in each *publicly visible state*; i.e., for each state outside of a public method or constructor's execution, and at the beginning and end of each public method's execution.[1] The figure's invariant just says that the value of `theStack` should never be `null`.

2.1.3 Method Specifications

Following the invariant are the specifications of the methods `pop`, `push`, and `top`. We describe the new aspects of these specifications below.

---

[1] In JML invariants also apply to non-public methods as well. The only exception is that a private method or constructor may be marked with the `helper` modifier; such methods cannot assume and do not need to establish the invariant.

2.1.3.1 The Assignable Clause

The use of the `assignable`[2] clauses in the behavioral specifications of `pop` and `push` is interesting (and another difference from Eiffel). These clauses give frame conditions [Borgida-Mylopoulos-Reiter95]. In JML, the frame condition given by a method's assignable clause only permits the method to assign to a location, *loc*, if:

- *loc* is mentioned in the method's `assignable` clause,
- *loc* is a member of a data group mentioned in the method's `assignable` clause (see Section 2.2 [Data Groups], page 19),
- *loc* was not allocated when the method started execution, or
- *loc* is local to the method (i.e., a local variable, including the method's formal parameters).

For example, `push`'s specification says that it may only assign to `theStack` (and locations in `theStack`'s data group). This allows `push` to assign to `theStack` (and the members of its data group), or to call some other method that makes such an assignment. Furthermore, `push` may assign to the formal parameter `x` itself, even though that location is not listed in the `assignable` clause, since `x` is local to the method. However, `push` may not assign to fields not mentioned in the `assignable` clause; in particular it may not assign to fields of its formal parameter `x`,[3] or call a method that makes such an assignment.

The design of JML is intended to allow tools to statically check the body of a method's implementation to determine whether its `assignable` clause is satisfied. This would be done by checking each assignment statement in the implementation to see if what is being assigned to is a location that some `assignable` clause permits. It is an error to assign to any other allocated, non-local location. However, to do this, a tool must conservatively track aliases and changes to objects containing the locations in question. Also, arrays can only be dynamically checked, in general.[4] Furthermore, JML will flag as an error a call to a method that would assign to locations that are not permitted by the calling method's `assignable` clause. It can do this using the `assignable` clause of the called method.

In JML, a location is *modified* by a method when it is allocated in both the pre-state of the method, reachable in the post-state, and has a value that is different in these two states. The *pre-state* of a method call is the state just after the method is called and parameters have been evaluated and passed, but before execution of the method's body. The *post-state* of a method call is the state just before the method returns or throws an exception; in JML we imagine that `\result` and information about exception results is recorded in the post-state.

Since modification only involves objects allocated in the pre-state, allocation of an object, using Java's `new` operator, does not itself cause any modification. Furthermore, since

[2] For historical reasons, one can also use the keyword `modifiable` as a synonym for `assignable`. Also, for compatibility with (older versions of) ESC/Java [Leino-etal00], in JML, one can also use the keyword `modifies` as a synonym for `assignable`. In the literature, the most common keyword for such a clause is `modifies`, and what JML calls the "assignable clause" is usually referred to as a "modifies clause". However, in JML, "assignable" most closely corresponds to the technical meaning, so we use that throughout this document. Users of JML may write whichever they prefer, and may mix them if they please.

[3] Assuming that `x` is not the same object as `this`!

[4] Thanks to Erik Poll for discussions on checking of assignable clauses.

the fields of new objects are locations that were not allocated when the method started execution, they may be assigned to freely.

The reason assignments to local variables are permitted by the assignable clause is that a JML specification takes the client's (i.e., the caller's) point of view. From the client's point of view, the local variables in a method are newly-allocated, and thus assignments to such variables are invisible to the client. Hence, in JML, it is an error to list the locations corresponding to formal parameters in the `assignable` clause. However, the locations corresponding to fields or array elements of such formal parameters can be sensibly mentioned in the `assignable` clause. Furthermore, when formal parameters are used in a postcondition, JML interprets these as meaning the value initially given to the formal in the pre-state, since assignments to the formals within the method do not matter to the client.

JML's interpretation of the assignable clause does not permit either temporary side effects or benevolent side effects. A method with a *temporary side effect* assigns to a location, does some work, and then assigns the original value back to that location. In JML, a method may not have temporary side effects on locations that it is not permitted to modify [Ruby-Leavens00]. A method has a *benevolent side effect* if it assigns to a location in a way that is not observable by clients. In JML, a method may not have benevolent side effects on locations that it is not permitted to modify [Leino95] [Leino95a].

Because JML's assignable clauses give permission to assign to locations, it is safe for clients to assume that only the listed locations (and locations of their data group members) may have their values modified. Because locations listed in the `assignable` clause are the only ones that can be modified, we often speak of what locations a method can "modify," instead of the more precise "can assign to."

What does the assignable clause say about the modification of locations? In particular, although the "location" for a model field or model variable cannot be directly assigned to in JML, its value is determined by the concrete fields and variables that it (ultimately) depends on, specifically the members of its data group. That is, a model field or variable can be modified by assignments to the concrete members of its data group (see Section 2.2 [Data Groups], page 19). Thus, a method's assignable clause only permits the method to modify a location if the location:

- is mentioned in the method's `assignable` clause,
- is a member of a data group mentioned in the `assignable` clause (see Section 2.2 [Data Groups], page 19),
- was not allocated when the method started execution, or
- is local to the method.

In the specification of `top`, the assignable clause says that a call to `top` that satisfies the precondition cannot assign to any locations. It does this by using the *store-ref* "`\nothing`." Unlike some formal specification languages (including Larch BISLs and older versions of JML), when the `assignable` clause is omitted in a heavyweight specification, the default *store-ref* for the assignable clause is `\everything`. Thus an omitted assignable clause in JML means that the method can assign to all locations (that could otherwise be assigned to by the method). Such an assignable clause plays havoc with formal reasoning, and thus if one cares about verification, one should give an assignable clause explicitly if the method is not pure (see Section 2.3.1 [Purity], page 29).

2.1.3.2 Old Values

When a method can modify some locations, they may have different values in the pre-state and post-state of a call. Often the post-condition must refer to the values held in both of these states. JML uses a notation similar to Eiffel's to refer to the pre-state value of a variable. In JML the syntax is `\old(`*E*`)`, where *E* is an expression. (Unlike Eiffel, we use parentheses following `\old` to delimit the expression to be evaluated in the pre-state explicitly. JML also uses backslashes (\\) to mark the keywords it uses in expressions; this avoids interfering with Java program identifiers, such as "`old`".)

The meaning of `\old(`*E*`)` is as if *E* were evaluated in the pre-state and that value is used in place of `\old(`*E*`)` in the assertion. It follows that, an expression like `\old(myVar).theStack` may not mean what is desired, since only the old value of `myVar` is saved; access to the field `theStack` is done in the post-state. If it is the field, `theStack`, not the variable, `myVar`, that is changing, then probably what is desired is `\old(myVar.theStack)`. To avoid such problems, it is good practice to have the expression *E* in `\old(`*E*`)` be such that its type is either the type of a primitive value, such as an `int`, or a type with immutable objects, such as `JMLObjectSequence`.

As another example, in `pop`'s postcondition the expression `\old(theStack.trailer())` has type `JMLObjectSequence`, so it is immutable. The value of `theStack.trailer()` is computed in the pre-state of the method.

2.1.3.3 Reference Semantics

Note also that, since `JMLObjectSequence` is a reference type, one must use `equals` instead of `==` to compare instances of this type for equality of values. For example, in the postcondition of the `pop` method, we use `equals` to compare `theStack` and `\old(theStack.trailer())`, as these may yield different objects. Using `==` would be a mistake, since it would only compare them for object identity.

As in Java itself, most types are reference types, and hence many expressions yield references (i.e., object identities or addresses), as opposed to primitive values. This means that `==`, except when used to compare pure values of primitive types such as `boolean` or `int`, is reference equality. As in Java, to get value equality for reference types one uses the `equals` method in assertions. For example, the predicate `myString == yourString`, is only true if the objects denoted by `myString` and `yourString` are the same object (i.e., if the names are aliases); to compare their values one must write `myString.equals(yourString)`.

2.1.3.4 Correct Implementation

The specification of `push` does not have a `requires` clause. This means that the method imposes no obligations on the caller. (The meaning of an omitted `requires` clause is that the method's precondition is `true`, which is satisfied by all states, and hence imposes no obligations on the caller.) This seems to imply that the implementation must provide a literally unbounded stack, which is surely impossible. We avoid this problem, by following Poetzsch-Heffter [Poetzsch-Heffter97] in releasing implementations from their obligations to fulfill the postcondition when Java runs out of storage. In general, a method specified with `normal_behavior` has a correct implementation if, whenever it is called in a state that satisfies its precondition, either

- the method terminates normally in a state that satisfies its postcondition, having assigned to only the locations permitted by its `assignable` clause, or

- Java signals an error, by throwing an exception that inherits from `java.lang.Error`.

We discuss the specification of methods with exceptions in the next subsection.

2.1.4 Models and Lightweight Specifications

In specifying existing code, one often does not want to introduce new model fields or think up new names for them. And sometimes, especially for fields with simple, atomic values, the field name itself is so "natural" that it would be difficult to think up a second good name for a model field that would be an abstraction of it. Thus JML provides two modifiers, `spec_public` and `spec_protected` that can used to make existing fields public or protected, for purposes of specification.

For example, consider the (lightweight) specification of the class `Point2D` below. In this specification the private fields, `x` and `y` are specified as `spec_public`, which allows them to be used in the public invariant clause and in the (implicitly public) specifications of the constructors and methods of `Point2D`.

```
package org.jmlspecs.samples.prelimdesign;

//@ model import org.jmlspecs.models.JMLDouble;

public class Point2D
{
  private /*@ spec_public @*/ double x = 0.0;
  private /*@ spec_public @*/ double y = 0.0;

  //@ public invariant !Double.isNaN(x);
  //@ public invariant !Double.isNaN(y);
  //@ public invariant !Double.isInfinite(x);
  //@ public invariant !Double.isInfinite(y);

  //@ ensures x == 0.0 && y == 0.0;
  public Point2D() { }

  /*@ requires !Double.isNaN(xc);
    @ requires !Double.isNaN(yc);
    @ requires !Double.isInfinite(xc);
    @ requires !Double.isInfinite(yc);
    @ assignable x, y;
    @ ensures x == xc && y == yc;
    @*/
  public Point2D(double xc, double yc) {
    x = xc;
    y = yc;
  }

  //@ ensures \result == x;
  public /*@ pure @*/ double getX() {
    return x;
```

```
    }

    //@ ensures \result == y;
    public /*@ pure @*/ double getY() {
      return y;
    }

    /*@ requires !Double.isNaN(x+dx);
      @ requires !Double.isInfinite(x+dx);
      @ assignable x;
      @ ensures JMLDouble.approximatelyEqualTo(x,
      @                         \old(x+dx), 1e-10);
      @*/
    public void moveX(double dx) {
      x += dx;
    }

    /*@ requires !Double.isNaN(y+dy);
      @ requires !Double.isInfinite(y+dy);
      @ assignable y;
      @ ensures JMLDouble.approximatelyEqualTo(y,
      @                         \old(y+dy), 1e-10);
      @*/
    public void moveY(double dy) {
      y += dy;
    }
  }
```

Note that these specifications would be illegal without the use of `spec_public`, since JML requires that public specifications only mention publicly-visible names (see Section 1.1 [Behavioral Interface Specification], page 1).

However, `spec_public` is more than just a way to change the visibility of a name for specification purposes. When applied to fields it can be considered to be shorthand for the declaration of a model field with the same name. That is, the declaration of `x` in `Point2D` can be thought of as equivalent to the following declarations, together with a rewrite of the Java code that uses `x` to use `_x` instead (where we assume `_x` is not used elsewhere).

```
    //@ public model int x;
    private int _x; //@ in x;
    //@ private represents x <- _x;
```

So in this way of thinking `spec_public` is not just an access modifier, but shorthand for declaration of a model field. This model field declaration is a commitment to readers that they can understand the specification using these model fields, even if the underlying private fields are changed, just as if the model field were declared explicitly. The model fields that are implicit allow such changes to be made without affecting the readers of the specification.

For example, suppose one wanted to change the implementation of `Point2D`, to use polar coordinates. To do that while keeping the public specification unchanged, one would declare the model fields `x` and `y` explicitly. One would then declare other fields for the polar and rectangular coordinates (and perhaps additional model fields as well). One would then also need to give explicit declarations that the new concrete fields are members of the model fields data groups, and give appropriate represents clauses. (See Section 2.2.2.1 [Data Groups and Represents Clauses], page 24, for more on data group membership and represents clauses.) All of this is exactly analogous to what is done implicitly in the the desugaring described above.

Similar remarks apply to `spec_protected`. The `spec_public` and `spec_protected` shorthands were borrowed from ESC/Java, but the desugaring described above is novel with JML.

2.2 Data Groups

In this subsection we present two example specifications. The two example specifications, `BoundedThing` and `BoundedStackInterface`, are used to describe how model (and concrete) fields can be related to one another, and how dependencies among them affect the meaning of the `assignable` clause. Along the way we also demonstrate how to specify methods that can throw exceptions and other features of JML.

2.2.1 Specification of BoundedThing

The specification in the file '`BoundedThing.java`', shown below, is an interface specification with a simple abstract model. In this case, there are two model fields `MAX_SIZE` and `size`.

```
package org.jmlspecs.samples.stacks;

public interface BoundedThing {

    //@ public model instance int MAX_SIZE;
    //@ public model instance int size;

    /*@ public instance invariant MAX_SIZE > 0;
        public instance invariant
                0 <= size && size <= MAX_SIZE;
        public instance constraint
                MAX_SIZE == \old(MAX_SIZE);
      @*/

    /*@  public normal_behavior
           ensures \result == MAX_SIZE;
      @*/
    public /*@ pure @*/ int getSizeLimit();

    /*@  public normal_behavior
           ensures \result <==> size == 0;
      @*/
    public /*@ pure @*/ boolean isEmpty();

    /*@  public normal_behavior
          ensures \result <==> size == MAX_SIZE;
```

```
      @*/
    public /*@ pure @*/ boolean isFull();

    /*@ also
         public behavior
           assignable \nothing;
           ensures \result instanceof BoundedThing
               && size == ((BoundedThing)\result).size;
           signals_only CloneNotSupportedException;
      @*/
    public Object clone ()
       throws CloneNotSupportedException;
}
```

After discussing the model fields, we describe the other parts of the specification below.

2.2.1.1 Model Fields in Interfaces

In the specification above, the fields `MAX_SIZE` and `size` are both declared using the modifier `instance`. Because of the use of the keyword `instance`, these fields are thus treated as normal model fields, i.e., as an instance variable in each object that implements this interface. By default, as in Java, fields are static in interfaces, and so if `instance` is omitted, the field declarations would be treated as class variables. The `instance` keyword tells the reader that the variable being declared is not static, but has a copy in each instance of a class that implements this interface.

Java does not allow non-static fields to be declared in interfaces. However, JML allows non-static model (and ghost) fields in interfaces when one uses `instance`. The reason for this extension is that such fields are essential for defining the abstract values and behavior of the objects being specified.[5]

In specifications of interfaces that extend or classes that implement this interface, these model fields are inherited. Thus, every object that has a type that is a subtype of the `BoundedThing` interface is thought of, abstractly, as having two fields, `MAX_SIZE` and `size`, both of type `int`.

2.2.1.2 Invariants and History Constraint

Three pieces of class-level specification come after the abstract model in the above specification.

The first two are `invariant` clauses. Writing several invariant clauses in a specification, like this is equivalent to writing one invariant clause which is their conjunction. Both of these invariants are instance invariants, because they use the `instance` modifier. By default, in interfaces, invariants and history constraints are static, unless marked with the `instance` modifier. Static invariants may only refer to static fields, while instance invariants can refer to both instance and static fields.

The first invariant in the figure says that in every publicly visible state, every reachable object that is a `BoundedThing` must have a positive `MAX_SIZE` field. The second invariant

[5] Furthermore, static model fields must have concrete implementations in the interfaces in which they are declared, if they are to have any representation at all. See Section 2.2.2.1 [Data Groups and Represents Clauses], page 24, for more on this subject.

says that, in each publicly visible state, every reachable object that is a `BoundedThing` must have a size field that is non-negative and less than or equal to `MAX_SIZE`.

Following the invariants is a history constraint [Liskov-Wing94]. Like the invariants, it uses the modifier `instance`, because it refers to instance fields. A history constraint is used to say how values can change between earlier and later publicly-visible states, such as a method's pre-state and its post-state. This prohibits subtype objects from making certain state changes, even if they implement more methods than are specified in a given class. The history constraint in the specification above says that the value of `MAX_SIZE` cannot change, since in every pre-state and post-state, its value in the post-state, written `MAX_SIZE`, must equal its value in the pre-state, written `\old(MAX_SIZE)`.

2.2.1.3 Details of the Method Specifications

Following the history constraint are the interfaces and specifications for four public methods. Notice that, if desired, the at-signs (`@`) may be omitted from the left sides of intermediate lines, as we do in this specification.

The use of `==` in the method specifications is okay, since in each case, the things being compared are primitive values, not references. The notation `<==>` can be read "if and only if". It has the same meaning for Boolean values as `==`, but has a lower precedence. Therefore, the expression "`\result <==> size == 0`" in the postcondition of the `isEmpty` method means the same thing as "`\result == (size == 0)`".

2.2.1.4 Adding to Method Specifications

The specification of the last method of `BoundedThing`, `clone`, is interesting. Note that it begins with the keyword `also`. This form is intended to tell the reader that the specification given is in addition to any specification that might have been given in the superclass `Object`, where `clone` is declared as a protected method. A form like this must be used whenever a specification is given for a method that overrides a method in a superclass, or that implements a method from an implemented interface.

2.2.1.5 Specifying Exceptional Behavior

The specification of `clone` also uses `behavior` instead of `normal_behavior`. In a specification that starts this way, one can describe not just the case where the execution returns normally, but also executions where exceptions are thrown. In such a specification, the conditions under which exceptions can be thrown can be described by the predicate in the `signals` clauses,[6] and the conditions under which the method may return without throwing an exception are described by the `ensures` clause. In this specification, the `clone` method may always throw the exception, because it only needs to make the predicate "`true`" true to do so. When the method returns normally, it must make the given postcondition true.

In JML, a `normal_behavior` specification can be thought of as a syntactic sugar for a `behavior` specification to which the following clause is added [Raghavan-Leavens05].

```
signals (java.lang.Exception) false;
```

This formalizes the idea that a method with a `normal_behavior` specification may not throw an exception when the specification's precondition is satisfied.

---

[6] The keyword "`exsures`" can also be used in place of `signals`.

JML also has a specification form `exceptional_behavior`, which can be used to specify when a method may not return normally. A specification that uses `exceptional_behavior` can be thought of as a syntactic sugar for a `behavior` specification to which the following clause is added [Raghavan-Leavens05].

```
ensures false;
```

This formalizes the idea that a method with an `exceptional_behavior` specification may not return normally when the specification's precondition is satisfied. Thus, when the precondition of such a specification case holds, some exception must be thrown (unless the execution encounters an error or is permitted to not return to the caller).

Since, in the specification of `clone`, we want to allow the implementation to make a choice between either returning normally or throwing an exception, and we do not wish to distinguish the preconditions under which each choice must be made, we cannot use either of the more specialized forms `normal_behavior` or `exceptional_behavior`. Thus the specification of `clone` demonstrates the somewhat unusual case when the more general form of a `behavior` specification is needed.

The specification of `clone` also illustrates the `signals_only` clause. The `signals_only` clause in the example says that the method may only throw an exception that is a subtype of `CloneNotSupportedException` when the exceptional behavior's precondition is true. This says the same thing as the following, more verbose, signals clause.

```
signals (Exception e) e instanceof CloneNotSupportedException;
```

The signals clause itself only describes what must be true when the exceptions it applies to are thrown; it does not constrain a method's behavior with respect to exceptions that are not subtypes of the exceptions named. For example, a `signals` clause of the form

```
signals (CloneNotSupportedException) true;
```

would only say that a `CloneNotSupportedException` can always be thrown; it would not prohibit other exceptions that are not subtypes of `CloneNotSupportedException` from being thrown. For example, if `clone` were specified with such a signals clause, then an implementation could legally throw a `NullPointerException`. To prevent such a possibility, in many cases it is preferable to use a `signals_only` clause to limit what exceptions may be thrown.

Finally note that in the specification of `clone`, the normal postcondition says that the result will be a `BoundedThing` and that its size will be the same as the model field `size`. The use of the cast in this postcondition is necessary, since the type of `\result` is `Object`. (This also adheres to our goal of using Java syntax and semantics to the extent possible.) Note also that the conjunct `\result instanceof BoundedThing` "protects" the next conjunct [Leavens-Wing97a] since if it is false the meaning of the cast does not matter.

2.2.2 Specification of BoundedStackInterface

The specification in the file '`BoundedStackInterface.java`' below gives an interface for bounded stacks that extends the interface for `BoundedThing`. Note that this specification can refer to the instance fields `MAX_SIZE` and `size` inherited from the `BoundedThing` interface.

```
package org.jmlspecs.samples.stacks;
//@ model import org.jmlspecs.models.*;
```

```
public interface BoundedStackInterface extends BoundedThing {
    //@ public initially theStack != null && theStack.isEmpty();
  /*@ public model instance JMLObjectSequence theStack;
    @                                            in size;
    @*/
  //@ public instance represents size <- theStack.int_length();
  /*@ public instance invariant theStack != null;
    @ public instance invariant_redundantly
    @                           theStack.int_length() <= MAX_SIZE;
    @ public instance invariant
    @         (\forall int i; 0 <= i && i < theStack.int_length();
    @                         theStack.itemAt(i) != null);
    @*/

  /*@   public normal_behavior
    @     requires !theStack.isEmpty();
    @     assignable size, theStack;
    @     ensures theStack.equals(\old(theStack.trailer()));
    @ also
    @   public exceptional_behavior
    @     requires theStack.isEmpty();
    @     assignable \nothing;
    @     signals_only BoundedStackException;
    @*/
  public void pop( ) throws BoundedStackException;

  /*@   public normal_behavior
    @     requires theStack.int_length() < MAX_SIZE && x != null;
    @     assignable size, theStack;
    @     ensures theStack.equals(\old(theStack.insertFront(x)));
    @     ensures_redundantly theStack != null && top() == x
    @              && theStack.int_length()
    @                     == \old(theStack.int_length()+1);
    @ also
    @   public exceptional_behavior
    @     requires theStack.int_length() >= MAX_SIZE || x == null;
    @     assignable \nothing;
    @     signals_only BoundedStackException, NullPointerException;
    @     signals (BoundedStackException)
    @                  theStack.int_length() == MAX_SIZE;
    @     signals (NullPointerException) x == null;
    @*/
  public void push(Object x )
         throws BoundedStackException, NullPointerException;

  /*@   public normal_behavior
    @     requires !theStack.isEmpty();
```

```
    @     ensures \result == theStack.first() && \result != null;
    @ also
    @   public exceptional_behavior
    @     requires theStack.isEmpty();
    @     signals_only BoundedStackException;
    @     signals (BoundedStackException e)
    @             \fresh(e) && e != null && e.getMessage() != null
    @           && e.getMessage().equals("empty stack");
    @*/
  public /*@ pure @*/ Object top( ) throws BoundedStackException;
}
```

The abstract model for `BoundedStackInterface` adds to the inherited model by declaring a model instance field named `theStack`. This field is typed as a `JMLObjectSequence`.

In the following we describe how the new model instance field, `theStack`, is related to `size` from `BoundedThing`. We also use this example to explain more JML features.

2.2.2.1 Data Groups and Represents Clauses

The `in` and `represents` clauses that follow the declaration of `theStack` are an important feature in modeling with layers of model fields.[7] They also play a crucial role in relating model fields to the concrete fields of objects, which can be considered to be the final layer of detail in a design.

When a model field is declared, a data group with the same name is automatically created; furthermore, this field is always a *member of* the group it creates. A *data group* is a set of fields (locations) referenced by a specific name, i.e., the name of the model field that created it [Leino98] [Leino-Poetzsch-Heffter-Zhou02].

When a data group (or field) is mentioned in the `assignable` clause for a method `M`, then all members (i.e., fields) in that group can be assigned to in the body of `M`. Fields can become a *member of* a data group through the data group clauses (i.e., the `in` and `maps-into` clauses) that come immediately after the field declaration, in this case the `in` clause. The `in` clause in `BoundedStackInterface` says that `theStack` is a member of the group created by the declaration of model field `size`; this means that `theStack` might change its value whenever `size` changes. However, another way of looking at this is that, if one wants to change `size`, this can be done by changing `theStack`. We also say that `theStack` is a *member of* `size`.

The *maps-into* clause is another way of adding members to a data group; it allows the fields of an object to be included in an existing data group. For example, if a field `F` is a reference or an array type, then the fields or array elements of `F` can be included in a data group using the *maps-into* clause. The following are examples.

```
protected ArrayList elems;
//@             maps elems.theList \into theStack;
protected java.lang.Object[] theItems;
//@             maps theItems[*] \into theStack;
```

[7] Of course, one could specify `BoundedStackInterface` without separating out the interface `BoundedThing`, and in that case, these layers would be unnecessary. We have made this separation partly to demonstrate more advanced features of JML, and partly to make the parts of the example smaller.

In the first example, the `maps-into` clause says that `theList` field of `elems` is a member of `theStack` data group. Field `elems` is a *concrete* field of the type (i.e., it is not a model field and thus is part of the implementation). This allows model field `theList` of `elems` to change when `theStack` changes. Since `theList` is a model field and data group, this also allows concrete fields of `elems` to change as `theStack` changes. Similarly, the second example says that the elements of the array, `theItems`, can change when `theStack` changes.

Data groups have the same visibility as the model field that declared it, i.e, public, protected, private, or package visibility. A field cannot be a member of a group that is less visible than it is. For example, a public field cannot be a member of a protected group.

The `in` and *maps-into* clauses are important in "loosening up" the `assignable` clause, for example to permit the fields of an object that implement the abstract model to be changed [Leino95] [Leino95a]. This "loosening up" also applies to model fields that are members of other groups. For example, since `theStack` is a member of `size`, whenever `size` is mentioned in an `assignable` clause, then `theStack` is implicitly allowed to be modified.[8] Thus it is only for rhetorical purposes that we mention both `size` and `theStack` in the assignable clauses of `pop` and `push`. Note, however, that just mentioning `theStack` would not permit `size` to be modified, because `size` is not a member of `theStack`'s group. Furthermore, it is redundant to mention `theStack` when `size` has already been mentioned (although this can help clarify the `assignable` clause, i.e., clarify which fields can be changed).

The `represents` clause in `BoundedStackInterface` says how the value of `size` is related to the value of `theStack`. It says that the value of `size` is `theStack.length()`.

A represents clause gives additional facts that can be used in reasoning about the specification. It serves the same purpose as an abstraction function in various proof methods for abstract data types (such as [Hoare72a]).

One can only use a represents clause to state facts about a field and its data group members. To state relationships among concrete data fields or on fields that are not related by a data group membership, one should use an invariant.

2.2.2.2 Redundant Specification

The second `invariant` clause that follows the `represents` clause in the specification of `BoundedStackInterface` above is our first example of checkable redundancy in a specification [Leavens-Baker99] [Tan94] [Tan95]. This concept is signaled in JML by the use of the suffix `_redundantly` on a keyword (as in `ensures_redundantly`). It says both that the stated property is specified to hold and that this property is believed to follow from the other properties of the specification. In this case the redundant invariant follows from the given invariant, the invariant inherited from the specification of `BoundedThing`, and the fact stated in the `represents` clause. Even though this invariant is redundant, it is sometimes helpful to state such properties, to bring them to the attention of the readers of the specification.

Checking that such claimed redundancies really do follow from other information is also a good way to make sure that what is being specified is really what is intended. Such

[8] Note that the permission to assign a field goes from the more abstract field to the one in its group (which in this case is also abstract). Müller points out that this direction is necessary for information hiding, because concrete fields are often hidden (e.g., they may be `private`), and as such cannot appear in public specifications, so the public specification has to mention the more abstract field, which give assignment rights to its members [Mueller02].

checks could be done manually, during reviews, or with the aid of a theorem prover. JML's runtime assertion checker can also check such redundant specifications, but, of course, can only find examples where they do not hold.

2.2.2.3 Multiple Specification Cases

After the redundant invariant of `BoundedStackInterface` are the specifications of the `pop`, `push`, and `top` methods. These are interesting for several new features that they present. Each of these has both a normal and exceptional behavior specified. The meaning of such multiple *specification cases* is that, when the precondition of one of them is satisfied, the rest of that specification case must also be obeyed.

A specification with several specification cases is shorthand for one in which the separate specifications are combined [Dhara-Leavens96] [Leavens97c] [Wing83] [Wills94]. The desugaring can be thought of as proceeding in two steps (see [Raghavan-Leavens05] for more details). First, the `public normal_behavior` and `public exceptional_behavior` cases are converted into `public behavior` specifications as explained above. This would produce a specification for `pop` as shown below. The use of `implies_that` introduces a redundant specification that can be used, as is done here, to point out consequences of the specification to the reader. In this case the specification in question is the one mentioned in the `refine` clause. Note that in the second specification case of the figure below, the default signals clause has been added. This clause was omitted from the original specification, since no particular details of the exception object were important to the specifier.

```
//@ refine "BoundedStackInterface.java";

public interface BoundedStackInterface extends BoundedThing {
  /*@ also
    @ implies_that
    @   public behavior
    @     requires !theStack.isEmpty();
    @     assignable size, theStack;
    @     ensures theStack.equals(\old(theStack.trailer()));
    @     signals (java.lang.Exception) false;
    @  also
    @   public behavior
    @     requires theStack.isEmpty();
    @     assignable \nothing;
    @     ensures false;
    @     signals_only BoundedStackException;
    @     signals (java.lang.Exception) true;
    @*/
  public void pop( ) throws BoundedStackException;
}
```

The second step of the desugaring is shown below. As can be seen from this example, `public behavior` specifications that are joined together using `also` have a precondition that is the disjunction of the preconditions of the combined specification cases. The `assignable` clause for the expanded specification is the union of all the assignable clauses for the cases. To compensate for this, the predicate `\not_assigned`, is used in the exceptional behavior specification cases to prohibit assignment to the locations (those in the data groups of `size` and `theStack`) that are now part of the assignable clause. The ensures clauses of the second desugaring step correspond to the ensures clauses for each specification case; they say that whenever the precondition for that specification case held in the pre-state, its postcondition must also hold. As can be seen in the specification below, in logic this is written using an implication between `\old` wrapped around the case's precondition and its postcondition. Having multiple ensures clauses is equivalent to writing a single ensures clause that has as its postcondition the conjunction of the given postconditions. Similarly, the signals clauses in the desugaring correspond to those in the given specification cases; as for the ensures clauses, each has a predicate that says that signaling that exception can only happen when the predicate in that case's precondition holds.

```
//@ refine "BoundedStackInterface.jml";
public interface BoundedStackInterface extends BoundedThing {
  /*@ also
    @ implies_that
    @   public behavior
    @     requires !theStack.isEmpty() || theStack.isEmpty();
    @     assignable size, theStack;
    @     ensures \old(!theStack.isEmpty())
    @              ==> theStack.equals(\old(theStack.trailer()));
    @     ensures \old(theStack.isEmpty()) ==>
    @                \not_assigned(size) && \not_assigned(theStack);
    @     signals_only BoundedStackException;
    @     signals (java.lang.Exception)
    @             \old(!theStack.isEmpty()) ==> false;
    @     signals (java.lang.Exception)
    @             \old(theStack.isEmpty()) ==>
    @              \not_assigned(size) && \not_assigned(theStack)
    @              && true;
    @*/
  public void pop( ) throws BoundedStackException;
}
```

In the file '`BoundedStackInterface.refines-java`' above, the precondition of `pop` reduces to `true`. However, the precondition shown is the general form of the expansion. Similar remarks apply to other predicates.

Finally, note how, as in the specification of `top`, one can specify more details about the exception object thrown. The exceptional behavior for `top` says that the exception object thrown, `e`, must be freshly allocated, non-null, and have the given message.

2.2.2.4 Pitfalls in Specifying Exceptions

A particularly interesting example of multiple specification cases occurs in the specification of the `BoundedStackInterface`'s `push` method. Like the other methods, this example has two specification cases; one of these is a `normal_behavior` and one is an `exceptional_behavior`. However, the exceptional behavior of `push` is interesting because it specifies more than one exception that may be thrown. The requires clause of the exceptional behavior says that an exception must be thrown when either the stack cannot grow larger, or when the argument `x` is null. The first signals clause says that, if a `BoundedStackException` is thrown, then the stack cannot grow larger, and the second signals clause says that, if a `NullPointerException` is thrown, then `x` must be null. The specification is written in this way because it may be that *both* conditions occur; when that is the case, the specification allows the implementation to choose (even nondeterministically) which exception is thrown.

Specifiers should be wary of such situations, where two different signals clauses may both apply simultaneously, because it is impossible in Java to throw more than one exception from a method call. Thus, for example, if the specification of `push` had been written as follows, it would not be implementable.[9] The problem is that both exceptional preconditions may be true, and in that case an implementation cannot throw an exception that is an instance of both a `BoundedStackException` and a `NullPointerException`.

```
/*@   public normal_behavior
  @     requires theStack.length() < MAX_SIZE && x != null;
  @     assignable size, theStack;
  @     ensures theStack.equals(\old(theStack.insertFront(x)));
  @     ensures_redundantly theStack != null && top() == x
  @              && theStack.length() == \old(theStack.length()+1);
  @ also
  @   public exceptional_behavior
  @     requires theStack.length() >= MAX_SIZE;
  @     assignable \nothing;
  @     signals (Exception e) e instanceof BoundedStackException;
  @ also                                   // this is wrong!
  @   public exceptional_behavior
  @     requires x == null;
  @     assignable \nothing;
  @     signals (Exception e) e instanceof NullPointerException;
  @*/
public void push(Object x )
       throws BoundedStackException, NullPointerException;
```

One could fix the example above by writing one of the requires clauses in the two exceptional behaviors to exclude the other, although this would make the specification deterministic about which exception would be thrown when both exceptional conditions occur. In general, it seems best to avoid this pitfall by writing signals clauses that do not exclude other exceptions from being thrown whenever there are states in which multiple exceptions may be thrown. That is, instead of using multiple `signals_only` clauses or using multiple `signals` clauses like:

```
signals (Exception e) e instanceof BoundedStackException;
```

which only allows a `BoundedStackException` to be thrown when the precondition is true, one can write a signals clause like:

```
signals (BoundedStackException);
```

which says nothing about what happens when other exceptions are thrown (see Section 2.2.1.5 [Specifying Exceptional Behavior], page 21 for more details).

[9] Thanks to Erik Poll for pointing this out.

2.2.2.5 Redundant Ensures Clauses

Finally, there is more redundancy in the specifications of `push` in the original specification of `BoundedStackInterface` above, which has a redundant `ensures` clause in its normal behavior. For an `ensures_redundantly` clause, what one checks is that the conjunction of the precondition, the meaning of the `assignable` clause, and the (non-redundant) postcondition together imply the redundant postcondition. It is interesting to note that, for `push`, the specifications for stacks written in Eiffel (see page 339 of [Meyer97]) expresses just what we specify in `push`'s redundant postcondition. This conveys strictly less information than the non-redundant postcondition for `push`'s normal behavior, since it says little about the elements of the stack.[10]

In summary, using types like `JMLObjectSequence` for modeling can help the specifier give more precise specifications. We describe more about such types in the next section.

2.3 Types For Modeling

JML comes with a suite of types with immutable objects and pure methods, that can be used for defining abstract models. These are found in the package `org.jmlspecs.models`, which includes both collection and non-collection types (such as `JMLInteger`) and a few auxiliary classes (such as exceptions and enumerators).

The collection types in this package can hold either objects or values; this distinction determines the notion of equality used on their elements and whether cloning is done on the elements. The object collections, such as `JMLObjectSet` and `JMLObjectBag`, use `==` and do not clone. The value collections, such as `JMLValueSet` and `JMLValueBag`, use `.equals` to compare elements, and clone the objects added to and returned from them. The objects in a value collection are representatives of equivalence classes (under `.equals`) of objects; their values matter, but not their object identities. By contrast an object container contains object identities, and the values in these objects do not matter.

Simple collection types include the set types, `JMLObjectSet` and `JMLValueSet`, and sequence types `JMLObjectSequence` and `JMLValueSequence`. The binary relation and map types can independently have objects in their domain or range. The binary relation types are named `JMLObjectToObjectRelation`, `JMLObjectToValueRelation`, and so on. For example, `JMLObjectToValueRelation` is a type of binary relations between objects (not cloned and compared using `==`) and values (which are cloned and compared using `.equals`). The four map types are similarly named according to the scheme `JML...To...Map`.

Users can also create their own types with pure methods for mathematical modeling if desired. Since pure methods may be used in assertions, they must be declared with the modifier `pure` and pass certain conservative checks that make sure there is no possibility of observable side-effects from their use. We discuss purity and give several examples of such types below.

2.3.1 Purity

We say a method is *pure* if it is either specified with the modifier `pure` or is a method that appears in the specification of a `pure` interface or class. Similarly, a constructor is pure if it is either specified with the modifier `pure` or appears in the specification of a `pure` class.

[10] Meyer's second specification and implementation of stacks (see page 349 of [Meyer97]) is no better in this respect, although, of course, the implementation does keep track of the elements properly.

A *pure method*, that is not a constructor, implicitly has a specification that does not allow any side-effects. That is, its specification has the clauses

```
diverges false;
assignable \nothing;
```

added to each specification case; if the method has no specification given explicitly, then these clauses are added as a lightweight specification. For this reason, if one is writing a pure method, it is not necessary to otherwise specify an assignable clause (see Section 2.1.3.1 [The Assignable Clause], page 13), although doing so may improve the specification's clarity.

A *pure constructor* has the clauses

```
diverges false;
assignable this.*;
```

added to each specification case; if the constructor has no specification given explicitly, then these clauses are added as a lightweight specification. This specification allows the constructor to assign to the non-static fields of the class in which it appears (including those inherited from its superclasses and model and ghost instance fields from the interfaces that it implements).

Implementations of pure methods and constructors will be checked to see that they meet these conditions; i.e., that pure methods do not assign to locations that exist in the pre-state, and that pure constructors only assign to pre-existing locations that are fields of the `this` object. To make such checking modular, some JML tools prohibit a pure method or constructor implementation from calling methods or constructors that are not pure. However, more sophisticated tools could more directly check the intended semantics [Salcianu-Rinard05].

A pure method or constructor must also be provably terminating. Although JML does not force users to make such proofs of termination, users writing pure methods and constructors are supposed to make pure methods total in the sense that whenever, a pure method is called it either returns normally or throws some exception. This is supposed to lessen the possibility that assertion evaluation could loop forever, help make pure methods more like mathematical functions, and help the runtime assertion checker. The runtime assertion checker turns exceptions into arbitrary values of the appropriate result type [Cheon03] [Cheon-Leavens05]; it cannot do anything with infinite loops.

Furthermore, a pure method is supposed to always either terminate normally or throw an exception, even for calls that do not satisfy its precondition. Static verification tools for JML should enforce this condition, by requiring a proof that a pure method implementation satisfies the following specification

```
private behavior
  requires true;
  diverges false;
  assignable \nothing;
```

(and similarly for constructors, except that the assignable clause becomes `assignable this.*;` for constructors).

However, this implicit verification condition is a specification, and is thus cannot be used in reasoning about calls to the method, even calls from within the class itself and recursive calls from within the implementation. For this reason we recommend writing the method or constructor specification in such a way that the effective precondition of the method is "true," making the proof of the above implicit verification condition trivial, and allowing the termination behavior of the implementation to be relied upon by all clients.

Recursion is permitted, both in the implementation of pure methods and the data structures they manipulate, and in the specifications of pure methods. When recursion is used in a specification, the proof of well-formedness for the specification involves the use of JML's `measured_by` clause.

Since a pure method may not go into an infinite loop, if it has a non-trivial precondition, it should throw an exception when its normal precondition is not met. This exceptional behavior does not have to be specified or programmed explicitly, but technically there is an obligation to meet the specification that the method never loops forever.

Furthermore, a pure method must be deterministic, in the sense that when called in a given state, it must always return the same value. Similarly a pure constructor should be deterministic in the sense that when called in a given state, it always initializes the object in the same way.

A pure method can be declared in any class or interface, and a pure constructor can be declared in any class. JML will specify the pure methods and constructors in the standard Java libraries as pure.

As a convenience, instead of writing `pure` on each method declared in a class and interface, one can use the modifier `pure` on classes and interfaces. This simply means that each non-static method and each constructor declared in such a class or interface is `pure`. Note that this does not mean that all methods inherited (but not declared in and hence not overridden in) the class or interface are pure. For example, every class inherits ultimately from `java.lang.Object`, which has some methods, such as `notify` and `notifyAll` that are manifestly not pure. Thus each class will have some methods that are not pure. Despite this, it is convenient to refer to classes and interfaces declared with the `pure` modifier as *pure*.

In JML the modifiers `model` and `pure` are orthogonal. (Recall something declared with the modifier `model` does not have to be implemented, and is used purely for specification purposes.) Therefore, one can have a model method that is not pure (these might be useful in JML's model programs); conversely, a Java method can be pure (and thus not a model method). Nevertheless, usually a model method (or constructor) should be pure, since there is no way to use non-pure methods in an assertion, and model methods cannot be used in normal Java code.

By the same reasoning, model classes should, in general, also be pure. Model classes cannot be used in normal Java code, and hence their methods are only useful in assertions (and JML's model programs). Hence it is typical, although not required, that a model class also be a pure class. We give some examples of pure interfaces, abstract classes, and classes below.

2.3.2 Money

The following example begins a specification of money that would be suitable for use in abstract models. Our specification is rather artificially broken up into pieces to allow each piece to have a specification that fits on a page. This organization is not necessarily something we would recommend, but it does give us a chance to illustrate more features of JML.

Consider first the interface `Money` specified below. The abstract model here is a single field of the primitive Java type `long`, which holds a number of pennies. Note that the declaration of this field, `pennies`, again uses the JML keyword `instance`.

```
package org.jmlspecs.samples.prelimdesign;

import org.jmlspecs.models.JMLType;

public /*@ pure @*/ interface Money extends JMLType
{
  //@ public model instance long pennies;

  //@ public instance constraint pennies == \old(pennies);

  /*@     public normal_behavior
    @       assignable \nothing;
    @       ensures \result == pennies / 100;
    @ for_example
    @     public normal_example
    @       requires pennies == 703;
    @       assignable \nothing;
    @       ensures \result == 7;
    @   also
    @     public normal_example
    @       requires pennies == 799;
    @       assignable \nothing;
    @       ensures \result == 7;
    @   also
    @     public normal_example
    @       requires pennies == -503;
    @       assignable \nothing;
    @       ensures \result == -5;
    @*/
  public long dollars();

  /*@   public normal_behavior
    @     assignable \nothing;
    @     ensures \result == pennies % 100;
    @ for_example
    @     requires pennies == 703;
    @     assignable \nothing;
    @     ensures \result == 3;
    @   also
    @     requires pennies == -503;
    @     assignable \nothing;
```

```
      @     ensures \result == -3;
      @*/
    public long cents();

    /*@ also
      @   public normal_behavior
      @     assignable \nothing;
      @     ensures \result
      @        <==> o2 instanceof Money
      @             && pennies == ((Money)o2).pennies;
      @*/
    public boolean equals(/*@ nullable @*/ Object o2);

    /*@ also
      @   public normal_behavior
      @     assignable \nothing;
      @     ensures \result instanceof Money
      @       && ((Money)\result).pennies == pennies;
      @*/
    public Object clone();
  }
```

This interface has a history constraint, which says that the number of pennies in an object cannot change.[11]

The following explain more aspects of JML related to the above specification.

2.3.2.1 Redundant Examples

The interesting aspect of `Money`'s method specifications is another kind of redundancy. This new form of redundancy is examples, which follow the keyword "`for_example`".

Individual examples are given by `normal_example` clauses (adapted from our previous work on Larch/C++ [Leavens96b] [Leavens-Baker99]). Any number of these[12] can be given in a specification. In the specification of `Money` above there are three normal examples given for `dollars` and two in the specification of `cents`.

The specification in each example should be such that:

- the example's precondition implies the precondition of the expanded meaning of the specified behaviors,
- the example's assignable clause specifies a subset of the locations that are assignable according to the expanded meaning of the specified behaviors, and
- assuming the example's assignable clause, the conjunction of:

---

[11] There is no use of `initially` in this interface, so data type induction cannot assume any particular starting value. But this is desirable, since if a particular starting value was specified, then by the history constraint, all objects would have that value.

[12] One may also give `exceptional_example` clauses, which are analogous to `exceptional_behavior` specifications, and `example` clauses, which are analogous to `behavior` specifications. There is also a lightweight form of example, this is similar to the `example` form, except that the introductory keywords "`public example`" are omitted.

- the example's precondition (wrapped by `\old()`),
- the precondition of the expanded meaning of the specified behaviors (also wrapped by `\old()`), and
- the postcondition of the expanded meaning of the specified behaviors

should be equivalent to the example's postcondition.

Requiring equivalence to the example's postcondition means that it can serve as a test oracle for the inputs described by the example's precondition. If there is only one specified `public normal_behavior` clause and if there are no preconditions and assignable clauses, then the example's postcondition should the equivalent to the conjunction of the example's precondition and the postcondition of the `public normal_behavior` specification. Typically, examples are concrete, and serve to make various rhetorical points about the use of the specification to the reader. (Exercise: check all the examples given!)

2.3.2.2 JMLType and Informal Predicates

The interface `Money` is specified to extend the interface `JMLType`. This interface is given below. Classes that implement this interface must have pure `equals` and `clone` methods with the specified behavior. The methods specified override methods in the class `Object`, and so they use the form of specification that begins with the keyword "`also`".

```
package org.jmlspecs.models;

/** Objects with a clone and equals method.
 * JMLObjectType and JMLValueType are refinements
 * for object and value containers (respectively).
 * @version $Revision: 1.20 $
 * @author Gary T. Leavens and Albert L. Baker.
 * @see JMLObjectType
 * @see JMLValueType
 */
//@ pure
public interface JMLType extends Cloneable, java.io.Serializable {

    /** Return a clone of this object. */
    /*@ also
      @   public normal_behavior
      @     ensures \result != null;
      @     ensures \result instanceof JMLType;
      @     ensures ((JMLType)\result).equals(this);
      @*/
    //@ implies_that
    /*@    ensures \result != null
      @        && \typeof(\result) <: \type(JMLType);
      @*/
    public /*@ pure @*/ Object clone();

    /** Test whether this object's value is equal to the given argument.
     */
    /*@ also
      @   public normal_behavior
      @     ensures \result ==>
      @             ob2 != null
      @             && (* ob2 is not distinguishable from this,
      @                   except by using mutation or == *);
      @ implies_that
      @   public normal_behavior
      @   {|
      @      requires ob2 != null && ob2 instanceof JMLType;
      @      ensures ((JMLType)ob2).equals(this) == \result;
      @    also
      @      requires ob2 == this;
      @      ensures \result <==> true;
      @   |}
      @*/
    public /*@ pure @*/ boolean equals(/*@ nullable @*/ Object ob2);

    /** Return a hash code for this object. */
    public /*@ pure @*/ int hashCode();
}
```

The specification of JMLType is noteworthy in its use of informal predicates [Leavens96b]. In JML these start with an open parenthesis and an asterisk (‘(*’) and continue until a matching asterisk and closing parenthesis (‘*)’). In the public specification of equals, the normal_behavior’s ensures clause uses an informal predicate as an escape from formality. The use of informal predicates avoids the delicate issues of saying formally what observable aliasing means, and what equality of values means in general.[13]

In the implies_that section of the specification of the equals method is a nested case analysis, between {| and |}. The meaning of this is that each pre- and postcondition pair has to be obeyed. The first of these nested pairs is essentially saying that equals has to be symmetric. The second of these is saying that it has to be reflexive.

The implies_that section of the clone method states some implications of the specification given that are useful for ESC/Java. These repeat, from the first part of clone’s specification, that the result must not be null, and that the result’s dynamic type, \typeof(\result), must be a subtype of (written <:) the type JMLType.

2.3.3 MoneyComparable and MoneyOps

The type Money lacks some useful operations. The extensions below provide specifications of comparison operations and arithmetic, respectively.

The specification in file ‘MoneyComparable.java’ is interesting because each of the specified preconditions protects the postcondition from undefinedness in the postcondition [Leavens-Wing97a]. For example, if the argument m2 in the greaterThan method were null, then the expression m2.pennies would not be defined.

```
package org.jmlspecs.samples.prelimdesign;

public /*@ pure @*/ interface MoneyComparable extends Money
{
  /*@ public normal_behavior
    @   requires m2 != null;
    @   assignable \nothing;
    @   ensures \result <==> pennies > m2.pennies;
    @*/
  public boolean greaterThan(Money m2);

  /*@ public normal_behavior
    @   requires m2 != null;
    @   assignable \nothing;
    @   ensures \result <==> pennies >= m2.pennies;
    @*/
  public boolean greaterThanOrEqualTo(Money m2);

  /*@ public normal_behavior
    @   requires m2 != null;
```

[13] *Observable aliasing* is a sharing relation between objects that can be detected by a program. Such a program, might, for example modify one object and read a changed value from the shared object. Formalizing this in general is beyond the scope of this paper, and probably beyond what JML can describe.

```
    @   assignable \nothing;
    @   ensures \result <==> pennies < m2.pennies;
    @*/
  public boolean lessThan(Money m2);

  /*@ public normal_behavior
    @   requires m2 != null;
    @   assignable \nothing;
    @   ensures \result <==> pennies <= m2.pennies;
    @*/
  public boolean lessThanOrEqualTo(Money m2);
}
```

The interface specified in the file 'MoneyOps.java' below extends the interface specified above. MoneyOps is interesting for the use of its pure model methods: inRange, can_add, and can_scaleBy. These methods cannot be invoked by Java programs; that is, they would not appear in the Java implementation. When, for example inRange is called in a predicate, it is equivalent to using some correct implementation of its specification. The specification of inRange also makes use of a local specification variable declaration, which follows the keyword "old". Such declarations allow one to abbreviate long expressions, or, to make rhetorical points by naming constants, as is done with epsilon. These old declarations are treated as locations that are initialized to the pre-state value of the given expression. Model methods can be normal (instance) methods as well as static (class) methods.

```
package org.jmlspecs.samples.prelimdesign;

public /*@ pure @*/ interface MoneyOps extends MoneyComparable
{
  /*@  public normal_behavior
    @     old double epsilon = 1.0;
    @     assignable \nothing;
    @     ensures \result
    @        <==> Long.MIN_VALUE + epsilon < d
    @             && d < Long.MAX_VALUE - epsilon;
    @ public model boolean inRange(double d);
    @
    @  public normal_behavior
    @     requires m2!= null;
    @     assignable \nothing;
    @     ensures \result
    @        <==> inRange((double) pennies + m2.pennies);
    @ public model boolean can_add(Money m2);
    @
    @  public normal_behavior
    @     ensures \result <==> inRange(factor * pennies);
    @ public model boolean can_scaleBy(double factor);
    @*/
```

```
    /*@   public normal_behavior
      @     old boolean can_add = can_add(m2); // FIXME: inline.
      @     requires m2 != null && can_add;
      @     assignable \nothing;
      @     ensures \result != null
      @          && \result.pennies == this.pennies + m2.pennies;
      @ for_example
      @   public normal_example
      @     requires this.pennies == 300 && m2.pennies == 400;
      @     assignable \nothing;
      @     ensures \result != null && \result.pennies == 700;
      @*/
    public MoneyOps plus(Money m2);

    /*@   public normal_behavior
      @     old boolean inRange = inRange((double) pennies - m2.pennies); // FIXME: inli
      @     requires m2 != null
      @           && inRange;
      @     assignable \nothing;
      @     ensures \result != null
      @          && \result.pennies == this.pennies - m2.pennies;
      @ for_example
      @   public normal_example
      @     requires this.pennies == 400 && m2.pennies == 300;
      @     assignable \nothing;
      @     ensures  \result != null && \result.pennies == 100;
      @*/
    public MoneyOps minus(Money m2);

    /*@   public normal_behavior
      @     requires can_scaleBy(factor);
      @     assignable \nothing;
      @     ensures \result != null
      @          && \result.pennies == (long)(factor * pennies);
      @ for_example
      @   public normal_example
      @     requires pennies == 400 && factor == 1.01;
      @     assignable \nothing;
      @     ensures \result != null && \result.pennies == 404;
      @*/
    public MoneyOps scaleBy(double factor);
  }
```

Note also that JML uses the Java semantics for mixed-type expressions. For example in the ensures clause of the above specification of `plus`, `m2.pennies` is implicitly coerced to a double-precision floating point number, as it would be in Java.

2.3.4 MoneyAC

The key to proofs that an implementation of a class or interface specification is correct lies in the use of `in`, `maps-into`, and `represents` clauses [Hoare72a] [Leino95].

Consider, for example, the abstract class specified in the file '`MoneyAC.java`' below. This class is abstract and has no constructors. The class declares a concrete field `numCents`, which is related to the model instance field `pennies` by the `represents` clause.[14] The represents clause states that the value of `pennies` is the value of `numCents`. This allows relatively trivial proofs of the correctness of the `dollars` and `cents` methods, and is key to the proofs of the other methods.

```
package org.jmlspecs.samples.prelimdesign;

public /*@ pure @*/ abstract class MoneyAC implements Money
{
  protected long numCents;
  //@                in pennies;

  //@ protected represents pennies <- numCents;

  /*@ protected constraint_redundantly
    @            numCents == \old(numCents); @*/

  public long dollars() {
    return numCents / 100;
  }

  public long cents() {
    return numCents % 100;
  }

  public boolean equals(/*@ nullable @*/ Object o2) {
    if (o2 instanceof Money) {
      Money m2 = (Money)o2;
      return numCents
             == (100 * m2.dollars() + m2.cents());
    } else {
      return false;
    }
  }

  public int hashCode() {
    return (int)numCents;
  }
```

[14] This `represents` clause is implicitly an instance, as opposed to a static, represents clause, because it appears in a class declaration.

```
    public Object clone() {
      return this;
    }
  }
```

2.3.5 MoneyComparableAC

The straightforward implementation of the pure abstract subclass `MoneyComparableAC` is given below. Besides extending the class `MoneyAC`, it implements the interface `MoneyComparable`. Note that the model and concrete fields are both inherited by this class.

```
package org.jmlspecs.samples.prelimdesign;

public /*@ pure @*/ abstract class MoneyComparableAC
    extends MoneyAC implements MoneyComparable
{
  protected static /*@ pure @*/
  long totalCents(Money m2)
  {
    long res = 100 * m2.dollars() + m2.cents();
    //@ assert res == m2.pennies;
    return res;
  }

  public boolean greaterThan(Money m2)
  {
    return numCents > totalCents(m2);
  }

  public boolean greaterThanOrEqualTo(Money m2)
  {
    return numCents >= totalCents(m2);
  }

  public boolean lessThan(Money m2)
  {
    return numCents < totalCents(m2);
  }

  public boolean lessThanOrEqualTo(Money m2)
  {
    return numCents <= totalCents(m2);
  }
}
```

An interesting feature of the class `MoneyComparableAC` is the protected static method named `totalCents`. For this method, we give its code with an embedded assertion, written following the keyword `assert`.[15]

Note that the model method, `inRange` is not implemented, and does not need to be implemented to make this class correctly implement the interface `MoneyComparable`.

2.3.6 USMoney

Finally, a concrete class implementation is given in the file '`USMoney.java`' shown below. The class `USMoney` implements the interface `MoneyOps`. Note that specifications as well as code are given for the constructors.

```
package org.jmlspecs.samples.prelimdesign;

public /*@ pure @*/ class USMoney
                extends MoneyComparableAC implements MoneyOps
{
  /*@   public normal_behavior
    @     assignable pennies;
    @     ensures pennies == cs;
    @ implies_that
    @   protected normal_behavior
    @     assignable pennies, numCents;
    @     ensures numCents == cs;
    @*/
  public USMoney(long cs)
  {
    numCents = cs;
  }

  /*@ public normal_behavior
    @   assignable pennies;
    @   ensures pennies == (long)(100.0 * amt);
    @   // ensures_redundantly (* pennies holds amt dollars *);
    @*/
  public USMoney(double amt)
  {
    numCents = (long)(100.0 * amt);
  }

  public MoneyOps plus(Money m2)
  {
```

[15] As of JDK 1.4, `assert` is also a reserved word in Java. One can thus write assert statements either in standard Java or in JML annotations. If one writes an assert statement as a JML annotation, all of the JML extensions to the Java expression syntax see Section 3.1 [Extensions to Java Expressions for Predicates], page 55 for the predicate can be used, but no side-effects are allowed in this predicate. Such a JML *assert-statement* may also refer to model and ghost variables. In a Java assert statement, i.e., in an *assert-statement* that is not in an annotation, one cannot use JML's extensions for assertions, because such assertions must compile with a Java compiler.

```
    return new USMoney(numCents + totalCents(m2));
  }

  public MoneyOps minus(Money m2)
  {
    return new USMoney(numCents - totalCents(m2));
  }

  public MoneyOps scaleBy(double factor)
  {
    return new USMoney(numCents * factor / 100.0);
  }

  public String toString()
  {
    return "$" + dollars() + "." + cents();
  }
}
```

The constructors each mention the fields that they initialize in their `assignable` clause. This is because the constructor's job is to initialize these fields. One can think of a `new` expression in Java as executing in two steps: allocating an object, and then calling the constructor. Thus the specification of a constructor needs to mention the fields that it can initialize in the `assignable` clause.

The first constructor's specification also illustrates that redundancy can also be used in an `assignable` clause. A redundant `assignable` clause follows if the meaning of the set of locations named is a subset of the ones denoted by the non-redundant clause for the same specification case. In this example the redundant assignable clause follows from the given assignable clause and the meaning of the `in` clause inherited from the superclass `MoneyAC`.

The second constructor above is noteworthy in that there is a redundant ensures clause that uses an informal predicate [Leavens96b]. In this instance, the informal predicate is used as a comment (which could also be used). Recall that informal predicates allow an escape from formality when one does not wish to give part of a specification in formal detail.

The `plus` and `minus` methods use `assume` statements; these are like assertions, but are intended to impose obligations on the callers [Back-Mikhajlova-vonWright98]. The main distinction between a `assume` statement and a `requires` clause is that the former is a statement and can be used within code. These may also be treated differently by different tools. For example, ESC/Java [Leino-etal00] will require callers to satisfy the requires clause of a method, but will not enforce the precondition if it is stated as an assumption.

2.4 Use of Pure Classes

Since `USMoney` is a pure class, it can be used to make models of other classes. An example is the abstract class specified in the file '`Account.jml`' below. The first model field in this class has the type `USMoney`, which was specified above. (Further explanation follows the specification below.)

```
package org.jmlspecs.samples.prelimdesign;
```

```
public class Account {
  //@ public model MoneyOps credit;
  //@ public model String accountOwner;
  /*@ public invariant accountOwner != null && credit != null
    @         && credit.greaterThanOrEqualTo(new USMoney(0)); @*/
  //@ public constraint accountOwner.equals(\old(accountOwner));

  /*@  public normal_behavior
    @    requires own != null && amt != null
    @          && (new USMoney(1)).lessThanOrEqualTo(amt);
    @    assignable credit, accountOwner;
    @    ensures credit.equals(amt) && accountOwner.equals(own);
    @*/
  public Account(MoneyOps amt, String own);

  /*@  public normal_behavior
    @    assignable \nothing;
    @    ensures \result.equals(credit);
    @*/
  public /*@ pure @*/ MoneyOps balance();

  /*@  public normal_behavior
    @    old boolean can_scale = credit.can_scaleBy(1.0 + rate);
    @    requires 0.0 <= rate && rate <= 1.0
    @          && can_scale;
    @    assignable credit;
    @    ensures credit.equals(\old(credit).scaleBy(1.0 + rate));
    @ for_example
    @  public normal_example
    @    requires rate == 0.05
    @          && (new USMoney(4000)).equals(credit);
    @    assignable credit;
    @    ensures credit.equals(new USMoney(4200));
    @*/
  public void payInterest(double rate);

  /*@  public normal_behavior
    @    old boolean can_add = credit.can_add(amt);
    @    requires amt != null
    @          && amt.greaterThanOrEqualTo(new USMoney(0))
    @          && can_add;
    @    assignable credit;
    @    ensures credit.equals(\old(credit).plus(amt));
    @ for_example
    @  public normal_example
    @    requires credit.equals(new USMoney(40000))
    @           && amt.equals(new USMoney(1));
```

```
      @    assignable credit;
      @    ensures credit.equals(new USMoney(40001));
      @*/
    public void deposit(MoneyOps amt);

    /*@  public normal_behavior
      @   requires amt != null
     @          && (new USMoney(0)).lessThanOrEqualTo(amt)
      @         && amt.lessThanOrEqualTo(credit);
      @    assignable credit;
      @    ensures credit.equals(\old(credit).minus(amt));
      @ for_example
      @  public normal_example
      @    requires credit.equals(new USMoney(40001))
      @          && amt.equals(new USMoney(40000));
      @    assignable credit;
      @    ensures credit.equals(new USMoney(1));
      @*/
    public void withdraw(MoneyOps amt);
  }
```

The specification of `Account` makes good use of examples. It also demonstrates the various ways of protecting predicates used in the specification from undefinedness [Leavens-Wing97a]. The principal concern here, as is often the case when using reference types in a model, is to protect against the model fields being `null`. As in Java, fields and variables of reference types can be `null`. In the specification of `Account`, the invariant states that these fields should not be null. Since implementations of public methods must preserve the invariants, one can think of the invariant as conjoined to the precondition and postcondition of each public method, and the postcondition of each public constructor. Hence, for example, method pre- and postconditions do not have to state that the fields are not null. However, often other parts of the specification must be written to allow the invariant to be preserved, or established by a constructor. For example, in the specification of `Account`'s constructor, this is done by requiring `amt` and `own` are not null, since, if they could be null, then the invariant and the postcondition could not be established.

2.5 Composition for Container Classes

The following specifications lead to the specification of a class `Digraph` (directed graph). This gives a more interesting example of how more complex models can be composed in JML from other classes. In this example we use model classes and the pure containers provided in the package `org.jmlspecs.models`.

2.5.1 NodeType

The file '`NodeType.java`' contains the specification of an interface `NodeType`. We also declare this interface to be `pure`, since we want to use its methods in the specifications of other classes. (This is appropriate, since all the methods of `NodeType` are side-effect free.)

```
package org.jmlspecs.samples.digraph;
```

```
import org.jmlspecs.models.*;

public /*@ pure @*/ interface NodeType extends JMLType {

  /*@ also
    @   public normal_behavior
    @       requires !(o instanceof NodeType);
    @       ensures \result == false;
    @*/
  public boolean equals(/*@ nullable @*/ Object o);

  public int hashCode();

  /*@ also
    @   public normal_behavior
    @     ensures \result instanceof NodeType
    @          && ((NodeType)\result).equals(this);
    @*/
  public Object clone();

}
```

2.5.2 ArcType

`ArcType` is specified as a pure class in the file '`ArcType.jml`' shown below. In theory, this class could have been declared with the `model` modifier, since it does not appear in the interface to `Digraph`. However, we specify it as a normal Java class for simplicity, and because model classes do not currently work in JML's runtime assertion checker. We declare `ArcType` to be a pure class so that its methods can be used in assertions. The two model fields for `ArcType`, `from` and `to`, are both of type `NodeType`. We specify the `equals` method so that two references to objects of type `ArcType` are equal if and only if they have equal values in the `from` and `to` model fields. Thus, `equals` is specified using `NodeType.equals`. We also specify that `ArcType` has a public `clone` method, fulfilling the obligations of a type that implements `JMLType`. `ArcType` must implement `JMLType` so that its objects can be placed in a `JMLValueSet`. We use such a set for one of the model fields of `Digraph`.

```
package org.jmlspecs.samples.digraph;

import org.jmlspecs.models.JMLType;

/*@ pure @*/ public class ArcType implements JMLType {

    //@ public model NodeType from;
    //@ public model NodeType to;
    //@ public invariant from != null && to != null;

    /*@ public normal_behavior
      @   requires from != null && to != null;
```

```
      @   assignable this.from, this.to;
      @   ensures this.from.equals(from)
      @        && this.to.equals(to);
      @*/
    public ArcType(NodeType from, NodeType to);

    /*@ also
      @   public normal_behavior
      @   {|
      @     requires o instanceof ArcType;
      @     ensures \result
      @        <==> ((ArcType)o).from.equals(from)
      @             && ((ArcType)o).to.equals(to);
      @   also
      @     requires !(o instanceof ArcType);
      @     ensures \result == false;
      @   |}
      @*/
    public boolean equals(/*@ nullable @*/ Object o);

    /*@ also
      @   public normal_behavior
      @     ensures \result instanceof ArcType
      @          && ((ArcType)\result).equals(this);
      @*/
    public Object clone();
}
```

The use of `also` in the specification of `ArcType`'s `equals` method is interesting. It separates two cases of the normal behavior for that method. This is equivalent to using two `public normal_behavior` clauses, one for each case. That is, when the argument is an instance of `ArcType`, the method must return true just when `this` and `o` have the same `from` and `to` fields. And when `o` is not an instance of `ArcType`, the `equals` method must return false.

2.5.3 Digraph

Finally, the specification of the class `Digraph` is given in the file '`Digraph.jml`' shown below. This specification demonstrates how to use container classes, like `JMLValueSet`, combined with appropriate invariants, to specify models that are compositions of other classes. In this specification, the container class `JMLValueSet` is used as the type of the model fields `nodes` and `arcs`. Since JML currently only works with a non-generic version of Java, the first invariant clause restricts `nodes` so that every object in `nodes` is, in fact, of type `NodeType`. Similarly, the next invariant clause we restrict `arcs` to be a set of `ArcType` objects. In both cases, since the type is `JMLValueSet`, membership is determined by the `equals` method for the type of the elements (rather than reference equality).

```
package org.jmlspecs.samples.digraph;
```

```
//@ model import org.jmlspecs.models.*;
public class Digraph {
 //@ public model JMLValueSet nodes;
 //@ public model JMLValueSet arcs;
 /*@ public invariant_redundantly nodes != null;
   @ public invariant (\forall JMLType n; nodes.has(n);
   @                          n instanceof NodeType);
   @ public invariant_redundantly arcs != null;
   @ public invariant (\forall JMLType a; arcs.has(a);
   @                          a instanceof ArcType);
   @ public invariant
   @      (\forall ArcType a; arcs.has(a);
   @           nodes.has(a.from) && nodes.has(a.to));
   @*/

 /*@  public normal_behavior
   @   assignable nodes, arcs;
   @   ensures nodes.isEmpty() && arcs.isEmpty();
   @*/
 public Digraph();

 /*@  public normal_behavior
   @   requires_redundantly n != null;
   @   assignable nodes;
   @   ensures nodes.equals(\old(nodes.insert(n)));
   @*/
 public void addNode(NodeType n);

 /*@  public normal_behavior
   @   requires unconnected(n);
   @   assignable nodes;
   @   ensures nodes.equals(\old(nodes.remove(n)));
   @*/
 public void removeNode(NodeType n);

 /*@  public normal_behavior
   @   requires_redundantly inFrom != null && inTo != null;
   @   requires nodes.has(inFrom) && nodes.has(inTo);
   @   assignable arcs;
   @   ensures arcs.equals(
   @            \old(arcs).insert(new ArcType(inFrom, inTo)));
   @*/
 public void addArc(NodeType inFrom, NodeType inTo);

 /*@  public normal_behavior
   @   requires_redundantly inFrom != null && inTo != null;
   @   requires nodes.has(inFrom) && nodes.has(inTo);
```

```
  @   assignable arcs;
  @   ensures arcs.equals(
  @            \old(arcs).remove(new ArcType(inFrom, inTo)));
  @*/
public void removeArc(NodeType inFrom, NodeType inTo);

/*@  public normal_behavior
  @   assignable \nothing;
  @   ensures \result == nodes.has(n);
  @*/
public /*@ pure @*/ boolean isNode(NodeType n);

/*@  public normal_behavior
  @   requires_redundantly inFrom != null && inTo != null;
  @   ensures \result == arcs.has(new ArcType(inFrom, inTo));
  @
  @*/
public /*@ pure @*/ boolean isArc(NodeType inFrom,
                                  NodeType inTo);

/*@  public normal_behavior
  @   requires nodes.has(start) && nodes.has(end);
  @   assignable \nothing;
  @   ensures \result
  @           == reachSet(new JMLValueSet(start)).has(end);
  @*/
public /*@ pure @*/ boolean isAPath(NodeType start,
                                    NodeType end);

/*@  public normal_behavior
  @   assignable \nothing;
  @   ensures \result <==>
  @             !(\exists ArcType a; arcs.has(a);
  @                    a.from.equals(n) || a.to.equals(n));
  @*/
public /*@ pure @*/ boolean unconnected(NodeType n);

/*@  public normal_behavior
  @   requires_redundantly nodeSet != null;
  @   requires (\forall JMLType o; nodeSet.has(o);
  @               o instanceof NodeType && nodes.has(o));
  @   {|
  @      assignable \nothing;
  @    also
  @      requires nodeSet.equals(OneMoreStep(nodeSet));
  @      ensures \result != null && \result.equals(nodeSet);
  @    also
```

```
      @      requires !nodeSet.equals(OneMoreStep(nodeSet));
      @      ensures \result != null
      @         && \result.equals(reachSet(OneMoreStep(nodeSet)));
      @   |}
      @ public pure model JMLValueSet reachSet(JMLValueSet nodeSet);
      @*/

    /*@  public normal_behavior
      @   requires_redundantly nodeSet != null;
      @   requires (\forall JMLType o; nodeSet.has(o);
      @              o instanceof NodeType && nodes.has(o));
      @      assignable \nothing;
      @   ensures_redundantly \result != null;
      @   ensures \result.equals(nodeSet.union(
      @        new JMLValueSet { NodeType n | nodes.has(n)
      @          && (\exists ArcType a; a != null && arcs.has(a);
      @                nodeSet.has(a.from) && n.equals(a.to))}));
      @ public pure model
      @ JMLValueSet OneMoreStep(JMLValueSet nodeSet);
      @*/
}
```

An interesting use of pure model methods appears at the end of the specification of `Digraph` in the pure model method `reachSet`. This method constructively defines the set of all nodes that are reachable from the nodes in the argument `nodeSet`. This specification uses a nested case analysis, between `{|` and `|}`. The meaning of this is again that each pre- and postcondition pair has to be obeyed, but by using nesting, one can avoid duplication of the requires clause that is found at the beginning of the specification. The `measured_by` clause is needed because this specification is recursive; the measure given allows one to describe a termination argument, and thus ensure that the specification is well-defined. This clause defines an integer-valued measure that must always be at least zero; furthermore, the measure for a call and recursive uses in the specification must strictly decrease [Owre-etal95]. The recursion in the specification builds up the entire set of reachable nodes by, for each recursive reference, adding the nodes that can be reached directly (via a single arc) from the nodes in `nodeSet`.

2.6 Behavioral Subtyping

As in Java, a subtype inherits members (fields and methods) from its supertypes. A subtype also inherits all the class level-specifications associated with fields and all method specifications for public and protected instance methods. This specification inheritance has the effect of making the subtype a behavioral subtype [Liskov-Wing94], in the sense that instances of the subtype obey the specifications its supertype(s) [Dhara-Leavens96] [Leavens-Weihl95].

Class-level specifications associated with fields include include invariants and history constraints (see Section 2.2.1.2 [Invariants and History Constraint], page 20), as well as `initially` clauses (see Section 2.1.1 [Model Fields], page 13) data group declarations (see Section 2.2 [Data Groups], page 19), and represents clauses (see Section 2.2.2.1 [Data Groups and Represents Clauses], page 24). Inheritance of invariants means that each supertype's invariants must also hold in the subtype. Similarly, every history constraint specified in each supertype must be obeyed in the subtype. And all initially clauses specified for supertype fields must also be obeyed in all subtypes. Fields declared in a supertype retain their data group membership when inherited. Their represents clauses are also inherited.

As in Java, private fields are inherited by a subtype but not visible to it. Similarly, default privacy (i.e., package visibility) fields are not accessible if the subtype is declared in a different package than the supertype declaring the field. As in Java, these fields are present in the objects of the subtype, but not accessible to code written in the subtype. In the same way, class level specifications associated with such fields must still be obeyed by objects of the subtype. Various restrictions to JML that ensure that this is always possible are being investigated [Ruby-Leavens00].

Specifications for instance methods are also inherited in the sense that public and protected specification cases must be obeyed by all overriding methods [Dhara-Leavens96] [Leavens97c]. This inheritance of method specifications ensures that a client's reasoning about a method call will still be valid, even if the method is overridden [America87] [America91] [Leavens-Weihl95], and thus that a subclass is a behavioral subtype of its supertypes [Dhara-Leavens96]. Note that private and default (package) visibility specification cases are not visible to subtypes, and hence do not have to be obeyed by them; not inheriting such specification cases does not cause clients reasoning problems, as these specification cases are not visible to clients making method calls (in general).[16] Furthermore, specifications are *not* inherited for constructors or for static methods, since they are not involved in dynamic dispatch.

Inheritance of method specifications can be thought of textually. For each instance method, m specified in a class C, one can imagine copying into the specification of m the public and protected specification cases for m given in all of C's ancestors and in all the interfaces C implements; these specification cases would be combined using `also` [Dhara-Leavens96] [Raghavan-Leavens05].[17] (This is the reason for the use of `also` at the beginning of specifications in overriding methods.) By the semantics of method combination using `also`, these behaviors must all be satisfied by the method, in addition to any explicitly specified behaviors.

For example, consider the class `PlusAccount`, specified in file '`PlusAccount.jml`' shown below. It is specified as a subclass of `Account` (see Section 2.4 [Use of Pure Classes], page 42). Thus it inherits the fields of `Account`, and `Account`'s public invariants, history constraints, and method specifications. (The specification of `Account` given above does not have any `protected` specification information.) Since it inherits the fields of its superclass, a textual copy of the method specification cases of `Account` would still be meaningful in the context of `PlusAccount`. Thinking of such textual copies works if one adds new (model) fields to specify the subclass and relates them to the existing ones. If instead one tried to respecify the fields of a supertype with invariants and history constraints that violated the (inherited) specification of that supertype, then the resulting specification would be contradictory, and hence not be correctly implementable.

[16] When such private and default visibility specification cases are visible to callers, they may only be used in verification of a method call if the call can be shown to be executing that method, as opposed to some override.

[17] However, textual copying shouldn't be taken literally; if a subclass declares a field that hides the fields of its superclass, renaming must be done to prevent name capture.

```
package org.jmlspecs.samples.prelimdesign;

public class PlusAccount extends Account {
  //@ public model MoneyOps savings, checking;  in credit;

  /*@ public represents credit \such_that
    @                  credit.equals(savings.plus(checking));
    @*/
  //@ public invariant savings != null && checking != null;
  /*@ public invariant_redundantly
    @           savings.plus(checking)
    @                  .greaterThanOrEqualTo(new USMoney(0));
    @*/

  /*@  public normal_behavior
    @    requires sav != null && chk != null && own != null
    @          && (new USMoney(1)).lessThanOrEqualTo(sav)
    @          && (new USMoney(1)).lessThanOrEqualTo(chk);
    @    assignable credit, owner;
    @    assignable_redundantly savings, checking;
    @    ensures savings.equals(sav) && checking.equals(chk)
    @             && owner.equals(own);
    @    ensures_redundantly credit.equals(sav.plus(chk));
    @*/
  public PlusAccount(MoneyOps sav, MoneyOps chk, String own);


  /*@ also
    @  public normal_behavior
    @    old boolean can_scale = credit.can_scaleBy(1.0 + rate);
    @    requires 0.0 <= rate && rate <= 1.0
    @          && can_scale;
    @    assignable credit, savings, checking;
    @    ensures checking.equals(
    @             \old(checking).scaleBy(1.0 + rate));
    @ for_example
    @  public normal_example
    @    requires rate == 0.05
    @          && checking.equals(new USMoney(2000));
    @    assignable credit, savings, checking;
    @    ensures checking.equals(new USMoney(2100));
    @*/
  public void payInterest(double rate);

  /*@ also
    @  public normal_behavior
    @    requires amt != null
```

```
  @            && (new USMoney(0)).lessThanOrEqualTo(amt)
  @            && amt.lessThanOrEqualTo(savings);
  @    assignable credit, savings;
  @    ensures savings.equals(\old(savings).minus(amt))
  @            && \not_modified(checking);
  @ also
  @  public normal_behavior
  @    requires amt != null
  @            && (new USMoney(0)).lessThanOrEqualTo(amt)
  @            && amt.lessThanOrEqualTo(credit)
  @            && amt.greaterThan(savings);
  @    assignable credit, savings, checking;
  @    ensures savings.equals(new USMoney(0))
  @           && checking.equals(
  @                \old(checking).minus(amt.minus(savings)));
  @ for_example
  @   public normal_example
  @    requires savings.equals(new USMoney(40001))
  @             && amt.equals(new USMoney(40000));
  @    assignable credit, savings, checking;
  @    ensures savings.equals(new USMoney(1))
  @            && \not_modified(checking);
  @  also
  @   public normal_example
  @    requires savings.equals(new USMoney(30001))
  @           && checking.equals(new USMoney(10000))
  @           && amt.equals(new USMoney(40000));
  @    assignable credit, savings, checking;
  @    ensures savings.equals(new USMoney(0))
  @           && checking.equals(new USMoney(1));
  @*/
public void withdraw(MoneyOps amt);

/*@ also
  @  public normal_behavior
  @    old boolean can_add = credit.can_add(amt);
  @    requires amt != null
  @           && amt.greaterThanOrEqualTo(new USMoney(0))
  @           && can_add;
  @    assignable credit, savings;
  @    ensures savings.equals(\old(savings).plus(amt))
  @            && \not_modified(checking);
  @ for_example
  @  public normal_example
  @    requires savings.equals(new USMoney(20000))
  @           && amt.equals(new USMoney(1));
  @    assignable credit, savings, checking;
```

```
  @    ensures savings.equals(new USMoney(20001));
  @*/
public void deposit(MoneyOps amt);

/*@    public normal_behavior
  @    old boolean can_add = credit.can_add(amt);
  @    requires amt != null
  @           && amt.greaterThanOrEqualTo(new USMoney(0))
  @           && can_add;
  @    assignable credit, checking;
  @    ensures checking.equals(\old(checking).plus(amt))
  @          && \not_modified(savings);
  @ for_example
  @  public normal_example
  @    requires checking.equals(new USMoney(20000))
  @           && amt.equals(new USMoney(1));
  @    assignable credit, checking;
  @    ensures checking.equals(new USMoney(20001));
  @*/
public void depositToChecking(MoneyOps amt);

/*@  public normal_behavior
  @    requires amt != null;
  @    {|
  @      requires (new USMoney(0)).lessThanOrEqualTo(amt)
  @             && amt.lessThanOrEqualTo(checking);
  @      assignable credit, checking;
  @      ensures checking.equals(\old(checking).minus(amt))
  @             && \not_modified(savings);
  @     also
  @      requires (new USMoney(0)).lessThanOrEqualTo(amt)
  @             && amt.lessThanOrEqualTo(credit)
  @             && amt.greaterThan(checking);
  @      assignable credit, checking, savings;
  @      ensures checking.equals(new USMoney(0))
  @          && savings.equals(
  @              \old(savings).minus(amt.minus(checking)));
  @    |}
  @ for_example
  @  public normal_example
  @    requires checking.equals(new USMoney(40001))
  @           && amt.equals(new USMoney(40000));
  @    assignable credit, checking;
  @    ensures checking.equals(new USMoney(1))
  @           && \not_modified(savings);
  @ also
  @  public normal_example
```

```
      @    requires savings.equals(new USMoney(30001))
      @           && checking.equals(new USMoney(10000))
      @           && amt.equals(new USMoney(40000));
      @    assignable credit, checking, savings;
      @    ensures checking.equals(new USMoney(0))
      @          && savings.equals(new USMoney(1));
      @*/
    public void payCheck(MoneyOps amt);
  }
```

Similarly, if one tried to respecify a method in a way that violated an (inherited) specification case, then the method would have to obey both specifications, and would not be correctly implementable. Thus, specification inheritance guarantees that all subtypes are behavioral subtypes in JML, and trying to avoid behavioral subtyping results in unimplementable specifications *Dhara-Leavens96*.

Note that in the represents clause below, instead of a left-facing arrow, `<-`, the connective "`\such_that`" is used to introduce a relationship predicate. This form of the represents clause allows one to specify abstraction relations, instead of abstraction functions.

3 Extensions to Java Expressions

JML makes extensions to the Java expression syntax for two purposes. The main set of extensions are used in predicates. But there are also some extensions used in *store-ref*s, which are themselves used in the `assignable` and `represents` clauses.

We give an overview of these extensions in this section. However, we only describe the most important and useful extensions here. See the *JML Reference Manual* [Leavens-etal-JMLRef] for more extensions and for more detail.

3.1 Extensions to Java Expressions for Predicates

The expressions that can be used as predicates in JML are an extension to the side-effect free Java expressions. Since predicates are required to be side-effect free, the following Java operators are *not* allowed within predicates:

- assignment (`=`), and the various assignment operators (such as `+=`, `-=`, etc.)
- all forms of increment and decrement operators (`++` and `--`),
- calls to methods that are not pure, and
- any use of operator `new` that would call a constructor that is not pure.

Furthermore, within method specifications that are not model programs, one cannot use `super` to call a pure superclass method, because it is confusing in combination with JML's specification inheritance.[1]

We allow the allocation of storage (e.g., using operator `new` and pure constructors) in predicates, because such storage can never be referred to after the evaluation of the predicate, and because such pure constructors have no side-effects other than initializing the new objects so created.

JML adds the following new syntax to the Java expression syntax, for use in predicates (see the *JML Reference Manual* [Leavens-etal-JMLRef] for syntactic details such as precedence):

- Informal descriptions, which look like

  ```
  (* some text describing a Boolean-valued predicate *)
  ```

  have type `boolean`. Their meaning is either `true` or `false`, but is entirely determined by the reader. Since informal descriptions are not-executable, they may be treated differently by different tools in different situations.
- `==>` and `<==` for logical implication and reverse implication. For example, the formula `raining ==> getsWet` is true if either `raining` is false or `getsWet` is true. The formula `getsWet <== raining` means the same thing. The `==>` operator associates to the right, but the `<==` operator associates to the left. The expressions on both sides of these operators must be of type `boolean`, and the type of the result is also `boolean`.
- `<==>` and `<=!=>` for logical equivalence and logical inequivalence, respectively. The expressions on either side of these operators must be of type `boolean`, and the type of

[1] Suppose A is the superclass of B, and B is the superclass of C. Suppose B's specification used `super` to call a method of A. The problem is that when this specification is inherited by C, if we imagine copying B's specification to C, then this use of *super* no longer refers to A, but to B. Thanks to Arnd Poetzsch-Heffter for pointing out this problem.

the result is also `boolean`. Note that `<==>` means the same thing as `==` for expressions of type `boolean`, and `<=!=>` means the same thing as `!=` for boolean expressions; however, `<==>` and `<=!=>` have a much lower precedence, and are also associative and symmetric.

- `\forall` and `\exists`, which are universal and existential quantifiers (respectively); for example,

  ```
  (\forall int i,j; 0 <= i && i < j && j < 10; a[i] < a[j])
  ```

  says that `a` is sorted at indexes between 0 and 9. The quantifiers range over all potential values of the variables declared which satisfy the *range* predicate, given between the semicolons (`;`). If the range predicate is omitted, it defaults to `true`. Since a quantifier quantifies over all potential values of the variables, when the variables declared are reference types, they may be null, or may refer to objects not constructed by the program; one should use a range predicate to eliminate such cases if they are not desired. The type of a universal and existential quantifier is `boolean`.

- `\max`, `\min`, `\product`, and `\sum`, which are generalized quantifiers that return the maximum, minimum, product, or sum of the values of the expressions given, where the variables satisfy the given range. The range predicate must be of type `boolean`. The expression in the body must be a built-in numeric type, such as `int` or `double`; the type of the quantified expression as a whole is the type of its body. The *body* of a quantified expression is the last top-level expression it contains; it is the expression following the range predicate, if there is one. As with the universal and existential quantifiers, if the range predicate is omitted, it defaults to `true`. For example, the following equations are all true (see chapter 3 of [Cohen90]):

  ```
  (\sum int i; 0 <= i && i < 5; i) == 0 + 1 + 2 + 3 + 4
  (\product int i; 0 < i && i < 5; i) == 1 * 2 * 3 * 4
  (\max int i; 0 <= i && i < 5; i) == 4
  (\min int i; 0 <= i && i < 5; i-1) == -1
  ```

  For computing the value of a sum or product, Java's arithmetic is used. The meaning thus depends on the type of the expression. For example, in Java, floating point numbers use the IEEE 754 standard, and thus when an overflow occurs, the appropriate positive or negative infinity is returned. However, Java integers wrap on overflow. Consider the following examples.

  ```
  (\product float f; 1.0e30f < f && f < 1.0e38f; f)
    == Float.POSITIVE_INFINITY

  (\sum int i; i == Integer.MAX_VALUE || i == 1; i)
    == Integer.MAX_VALUE + 1
    == Integer.MIN_VALUE
  ```

  When the range predicate is not satisfiable, the sum is 0 and the product is 1; for example:

  ```
  (\sum int i; false; i) == 0
  (\product double d; false; d*d) == 1.0
  ```

  When the range predicate is not satisfiable for `\max` the result is the smallest number with the type of the expression in the body; for floating point numbers, negative infinity is used. Similarly, when the range predicate is not satisfiable for `\min`, the result is the largest number with the type of the expression in the body.

- `\num_of`, which is "numerical quantifier." It returns the number of values for its variables for which the range and the expression in its body are true. Both the range predicate and the body must have type `boolean`, and the entire quantified expression has type `long`. The meaning of this quantifier is defined by the following equation (see p. 57 of [Cohen90]).

```
(\num_of T x; R(x); P(x)) == (\sum T x; R(x) && P(x); 1L)
```

- Set comprehensions, which can be used to succinctly define sets; for example, the following is the `JMLObjectSet` that is the subset of non-null `Integer` objects found in the set `myIntSet` whose values are between 0 and 10, inclusive.

```
new JMLObjectSet {Integer i | myIntSet.has(i)
                              && i != null && 0 <= i.getInteger()
                              && i.getInteger() <= 10 }
```

  The syntax of JML (see the *JML Reference Manual* [Leavens-etal-JMLRef] for details) limits set comprehensions so that following the vertical bar ('`|`') is always an invocation of the `has` method of some set on the variable declared. (This restriction is used to avoid Russell's paradox [Whitehead-Russell25].) In practice, one either starts from some relevant set at hand, or one can start from the sets containing the objects of primitive types found in `org.jmlspecs.models.JMLModelObjectSet` and (in the same Java package) `JMLModelValueSet`. The type of such an expression is the type named following `new`, which must be `JMLObjectSet` or `JMLValueSet`.

- `\elemtype`, which returns the most-specific static type shared by all elements of its array argument [Leino-Nelson-Saxe00]. For example, `\elemtype(\type(int[]))` is `\type(int)`. The argument to `\elemtype` must be an expression of type `\TYPE`, which JML considers to be the same as `java.lang.Class`, and its result also has type `\TYPE`. If the argument is not an array type, the result is `null`.

- `\fresh`, which asserts that objects were freshly allocated; for example, `\fresh(x,y)` asserts that `x` and `y` are not null and that the objects bound to these identifiers were not allocated in the pre-state. The arguments to `\fresh` can have any reference type, and the type of the overall expression is `boolean`.[2]

- `\nonnullelements`, which can be used to assert that an array and its elements are all non-null. For example, `\nonnullelements(myArray)`, is equivalent to [Leino-Nelson-Saxe00]

```
myArray != null &&
(\forall int i; 0 <= i && i < myArray.length;
                myArray[i] != null)
```

- `\old`, which can be used to refer to values in the pre-state; e.g., `\old(myPoint.x)` is the value of the `x` field of the object `myPoint` in the pre-state. The type of such an expression is the type of the expression it contains; for example the type of `\old(myPoint.x)` is the type of `myPoint.x`. The keyword `\old` can only be used in an *ensures-clause*, a *signals-clause*, or a *history-constraint*; it cannot be used, for example, in preconditions.

- `\result`, which, in an `ensures` clause is the value or object that is being returned by a method. Its type is the return type of the method; hence it is a type error to

[2] Note that it is wrong to use `\fresh(this)` in the specification of a constructor, because Java's `new` operator allocates storage for the object; the constructor's job is just to initialize that storage.

use `\result` in a void method or in a constructor. The keyword `\result` can only be used in an *ensures-clause*; it cannot be used, for example, in preconditions or in signals clauses.

- `\typeof`, which returns the most-specific dynamic type of an expression's value [Leino-Nelson-Saxe00]. The meaning of `\typeof(`*E*`)` is unspecified if *E* is null. If *E* has a static type that is a reference type, then `\typeof(`*E*`)` means the same thing as *E*`.getClass()`. For example, if `c` is a variable of static type `Collection` that holds an object of class `HashSet`, then `\typeof(c)` is `HashSet.class`, which is the same thing as `\type(HashSet)`. If *E* has a static type that is not a reference type, then `\typeof(`*E*`)` means the instance of `java.lang.Class` that represents its static type. For example, `\typeof(true)` is `Boolean.TYPE`, which is the same as `\type(boolean)`. Thus an expression of the form `\typeof(`*E*`)` has type `\TYPE`, which JML considers to be the same as `java.lang.Class`.
- `<:`, which compares two reference types and returns true when the type on the left is a subtype of the type on the right [Leino-Nelson-Saxe00]. Although the notation might suggest otherwise, this operator is also reflexive; a type will compare as `<:` with itself. In an expression of the form *E1* `<:` *E2*, both *E1* and *E2* must have type `\TYPE`; since in JML `\TYPE` is the same as `java.lang.Class` the expression *E1* `<:` *E2* means the same thing as the expression *E2*`.isAssignableFrom(`*E1*`)`.
- `\type`, which can be used to mark types in expressions. An expression of the form `\type(T)` has the type `\TYPE`. Since in JML `\TYPE` is the same as `java.lang.Class`, an expression of the form `\type(`*T*`)` means the same thing as *T*`.class`. For example, in

  ```
  \typeof(myObj) <: \type(PlusAccount)
  ```

  the use of `\type(PlusAccount)` is used to introduce the type `PlusAccount` into this expression context.

To avoid referring to the value of uninitialized locations, a constructor's precondition can only refer to locations in the object being constructed that are not assignable. This allows a constructor to refer to instance fields of the object being constructed if they are not made assignable by the constructor's assignable clause, for example, if they are declared with initializers. In particular, the precondition of a constructor may not mention a "blank final" instance variable that it must assign.

Since we are using Java expressions for predicates, there are some additional problems in mathematical modeling. We are excluding the possibility of side-effects by limiting the syntax of predicates, and by using type checking [Gifford-Lucassen86] [Lucassen87] [Lucassen-Gifford88] [Nielson-Nielson-Amtoft97] [Talpin-Jouvelot94] [Wright92] to make sure that only pure methods and constructors may be called in predicates.

Exceptions in expressions are particularly important, since they may arise in type casts. JML deals with exceptions by having the evaluation of predicates substitute an arbitrary expressible value of the normal result type when an exception is thrown during evaluation. When the expression's result type is a reference type, an implementation would have to return `null` if an exception is thrown while executing such a predicate. This corresponds to a mathematical model in which partial functions are mathematically modeled by underspecified total functions [Gries-Schneider95]. However, tools sometimes only approximate this semantics. In tools, instead of fully catching exceptions for all subexpressions, many tools only catch exceptions for the smallest boolean-valued subexpression that may throw an exception (and for entire expressions used in JML's *measured-clause* and *variant-function*).

JML will check that errors (i.e., exceptions that inherit from Error) are not explicitly thrown by pure methods. This means that they can be ignored during mathematical modeling. When executing predicates, errors will cause run-time errors.

3.2 Extensions to Java Expressions for Store-Refs

The grammatical production *store-ref* (see the *JML Reference Manual* [Leavens-etal-JMLRef] for the exact syntax) is used to name locations in the `assignable` and `represents` clauses. A *store-ref* names a location, not an object; a location is either a field of an object, or an array element. Besides the Java syntax of names and field and array references, JML supports the following syntax for *store-ref*s. See the *JML Reference Manual* [Leavens-etal-JMLRef] for more details on the syntax.

- Array ranges, of the form *A*`[`*E1* `..` *E2*`]`, denote the locations in the array *A* between the value of *E1* and the value of *E2* (inclusive). For example, the clause

  ```
  assignable myArray[3 .. 5]
  ```

  can be thought of an abbreviation for the following.

  ```
  assignable myArray[3], myArray[4], myArray[5]
  ```

- One can also name all the indexes in an array *A* by writing, *A*`[*]`, which is shorthand for *A*`[0 ..` *A*`.length-1]`.
- Two notations allow one to refer to the fields in some particular object.
  - The syntax *x*`.*` names all of the non-static fields of the object referred to by *x*. For example, if `p` is a `Point` object with two fields, `x` and `y` of type `BigInteger`, then `p.*` names the fields `p.x` and `p.y`. Notice that the fields of the `BigInteger` objects are not named. Also, `p.*.*` is not allowed.
  - If `a` is an array of type `Rocket []`, then the *store-ref* `a[*].*` means all of the non-static fields of each `Rocket` object referred to by the elements of array `a`.

4 Conclusions

One area of future work for JML is concurrency. Some recent work by Rodriguez *et al.* [Rodriguez-etal05] has investigated the use of atomicity for specifying multi-threaded Java programs. However, these ideas are not yet implemented in most of the JML tools, and their use has not been fully explored.

JML has also been used as a research vehicle in a wide variety of other studies various papers on these ideas can be found through the JML web page '`http://www.jmlspecs.org/`'.

JML is an expressive behavioral interface specification language for Java. It combines the best features of the Eiffel and Larch approaches to specification. It allows one to write specifications that are quite precise and detailed, but also allows one to write lightweight specifications. It has examples and other forms of redundancy to allow for debugging specifications and for making rhetorical points. It supports behavioral subtyping by specification inheritance.

More information on JML, including software to aid in working with JML specifications, can be obtained from '`http://www.jmlspecs.org/`'. The JML web site also includes an up-to-date version of this document with a table of contents and an index.

Acknowledgments

The work of Leavens and Ruby was supported in part by a grant from Rockwell International Corporation and by NSF grant CCR-9503168. Work on JML by Leavens, Baker, and Ruby was also supported in part by NSF grant CCR-9803843. Work on JML by Leavens, Ruby, and others is supported in part by NSF grants CCR-0097907,CCR-0113181, CCF-0428078, and CCF-0429567.

Many people have helped with the semantics and design of JML, and on this document. Thanks to Yoonsik Cheon, David Cok, Bart Jacobs, Rustan Leino, Peter Müller, Erik Poll, Arnd Poetzsch-Heffter, and Joachim van den Berg, for many discussions about the semantics of JML specifications. Thanks to Raymie Stata for spear-heading an effort at Compaq SRC to unify JML and ESC/Java, and to Rustan and Raymie for many interesting ideas and discussions that have profoundly influenced JML. For comments on earlier drafts and discussions about JML thanks to Yoonsik, Bart, Rustan, Peter, Eric, Joachim, Raymie, Abhay Bhorkar, Patrice Chalin, Curtis Clifton, John Boyland, Martin Büchi, Peter Chan, David Cok, Gary Daugherty, Jan Docxx, Marko van Dooren, Stephen Edwards, Michael Ernst, Arthur Fleck, Karl Hoech, Marieke Huisman, Anand Ganapathy, Doug Lea, Claude Marche, Kristof Mertens, Yogy Namara, Sevtap Oltes, Arnd Poetzsch-Heffter, Jim Potts, Arun Raghavan, Alexandru D. Salcianu, Jim Saxe, Tammy Scherbring, Tim Wahls, Wolfgang Weck, and others we may have forgotten. Thanks to David Cok, Yoonsik Cheon, Curtis Clifton, Patrice Chalin, Abhay Bhorkar, Kristina Boysen, Tongjie Chen, Kui Dai, Werner Dietel, Marko van Dooren, Anand Ganapathy, Yogy Namara, Todd Millstein, Arun Raghavan, Frederic Rioux, Roy Tan, and Hao Xi for their work on the JML checker and tools used to check and manipulate the specifications in this document. Thanks to Katie Becker, Kristina Boysen, Brandon Shilling, Elizabeth Seagren, Ajani Thomas, and Arthur Thomas for help with case studies and specifications in JML. Thanks to David Cok, Joe Kiniry, Yoonsik Cheon, Kristina Boysen, Curtis Clifton, Judy Chan Wai Ting, Peter Chan, Marko van Dooren, Kui Dai, Fermin da Costa Gomez, Joseph Kiniry, Roy Patrick Tan, and Julien Vermillard for bug reports about JML tools. Thanks to the students in 22C:181 at the University of Iowa in Spring 2001, and in Com S 362 at Iowa State University for suggestions and comments about JML.

Appendix A Specification Case Defaults

As noted above (see Section 1.2 [Lightweight Specifications], page 5), specifications in JML do not need to be as detailed as most of the examples given in this document. If a *spec-case* does not use one of the behavior keywords (`behavior`, `normal_behavior`, or `exceptional_behavior`), or if an *example* does not use one of the example keywords (`example`, `normal_example`, `exceptional_example`), then it is called a *lightweight* specification or example. Otherwise it is a *heavyweight* specification or example.

When the various clauses of a *spec-case* or *example* are omitted, they have the defaults given in the table below. The table distinguishes between lightweight and heavyweight specifications and examples. In most cases the default for the lightweight form is that no assumption is made about the omitted clause. There are only two exceptions to this rule. The first is that for an omitted `diverges` clause defaults to `false`. The second is that an omitted `signals_only` has a default that depends on the exceptions declared in the method.

```
                             Default
Omitted clause  lightweight                     heavyweight

----------------------------------------------------------------
requires        \not_specified                  true
diverges        false                           false
measured_by     \not_specified                  \not_specified
assignable      \not_specified                  \everything
when            \not_specified                  true
working_space   \not_specified                  \not_specified
duration        \not_specified                  \not_specified
ensures         \not_specified                  true
signals_only    see text                        see text
signals         \not_specified                  (Exception) true
```

However, in a heavyweight specification or example, the specifier is assumed to be giving a complete specification or example. Therefore, in a heavyweight specification the meaning of an omitted clause is given a definite default. For example, the meaning of an omitted `assignable` clause is that all locations (that can otherwise be legally assigned to) can be assigned. Furthermore, in a non-lightweight specification, the meaning of an omitted diverges clause is that the method may not diverge in that case. (The `diverges` clause is almost always omitted; it can be used to say what should be true, of the pre-state, when the specification is allowed to loop forever or signal an error.)

The defaults for omitted `signals_only` and `signals` clauses do not apply to `normal_behavior` specification cases, since `normal_behavior` specification cases cannot contain these clauses.

The default for the `signals_only` clause is the same in both heavyweight and lightweight specifications, but depends on the list of exceptions declared in the method. If the method declares no exceptions, then the default clause is `signals_only \nothing;` (which means that the method cannot throw any exceptions). However, if the method header declares that the method may throw exceptions *DE1*, . . ., *DEn*, then the default `signals_only` clause is as follows.

```
signals_only DE1, ..., DEn
```

This default applies whenever the `signals_only` clause is omitted from a specification case, even if the method has a `signals` clause. If this default is too strong, the effect can be changed by either writing an explicit `signals_only` clause, or by declaring further exceptions in the method's header. Note that although Java allows runtime exceptions (subtypes of `java.lang.RuntimeException`) to be thrown without being declared in a method's header, JML does not take make a special case for these.

A completely omitted specification is taken to be a lightweight specification. If the default (zero-argument) constructor of a class is omitted because its code is omitted, then its specification defaults to an assignable clause that allows all the locations that the default (zero-argument) constructor of its superclass assigns — in essence a copy of the superclass's default constructor's assignable clause. If some other frame is desired, then one has to write the specification, or at least the code, explicitly.

A method or constructor with code present has a *completely omitted* specification if it has no *specification-cases* and does not use annotations like `non_null` or `pure` that add implicit specifications.

If a method or constructor has code, has a completely omitted specification, and does not override another method, then its meaning is taken as the lightweight specification `diverges \not_specified;`. Thus, its meaning can be read from the lightweight column of table above, except that the diverges clause is not given its usual default. This is done so that the default specification when no specification is given truly says nothing about the method's behavior. However, if a method with code and a completely omitted specification overrides some other method, then its meaning is taken to be the lightweight specification `also requires false;`. This somewhat counter-intuitive specification is the unit under specification conjunction with `also`; it is used so as not to change the meaning of the inherited specification.

If the code is annotated with keywords like `non_null` or `pure` that add implicit specifications, then these implicit specifications are used instead of the default. Code with such annotations is considered to have an implicit specification.

It is intended that the meaning of `\not_specified` may vary between different uses of a JML specification. For example, a static checker might treat a `requires` clause that is `\not_specified` as if it were `true`, while a verification logic might treat it as if it were `false`. However, a reasonable default for the interpretation for `\not_specified` in a lightweight specification is the most liberal possible (i.e., the one that permits the most correct implementations); this is generally the same as the heavyweight default, except for the `diverges` clause (where the most liberal interpretation would be `true`).

Note that specification statements (see the *JML Reference manual* [Leavens-etal-JMLRef] for details) cannot be lightweight. In addition, a *spec-statement* can specify abrupt termination. The additional clauses possible in a *spec-statement* have the following defaults. These are not liberally interpreted, but instead prohibit the statement from having abrupt behavior by default.

```
               Default
Omitted clause (heavyweight)
----------------------------
continues      false
```

`breaks`	`false`
`returns`	`false`

Bibliography

[Arnold-Gosling-Holmes00]
Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language Third Edition.* The Java Series. Addison-Wesley, Reading, MA, 2000.

[America87]
Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, New York, NY, June 1987. Springer-Verlag. *Lecture Notes in Computer Science*, volume 276.

[America91]
Pierre America. Designing an object-oriented programming language with behavioural subtyping. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands, May/June 1990*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, New York, NY, 1991.

[Back88] R. J. R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25(6):593–624, August 1988.

[Back-vonWright89a]
R. J. R. Back and J. von Wright. Refinement Calculus, Part I: Sequential Nondeterministic Programs. In J. W. de Bakker, et al, (eds.), *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop*, Mook, The Netherlands, May/June 1989, pages 42-66. Volume 430 of *Lecure Notes Computer Science*, Spring-Verlag, 1989.

[Back-Mikhajlova-vonWright98]
Ralph Back, Anna Mikhajlova, and Joakim von Wright. Modeling component environments and interactive programs using iterative choice. Technical Report 200, Turku Centre for Computer Science, September 1998.
'`http://www.tucs.abo.fi/publications/techreports/TR200.html`'.

[Back-vonWright98]
Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* Springer-Verlag, 1998.

[Beck-Gamm98]
Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[Buechi-Weck00]
Martin Büchi and Wolfgang Weck. The Greybox Approach: When Blackbox Specifications Hide Too Much. Technical Report 297, Turku Centre for Computer Science, August 1999.
'`http://www.tucs.abo.fi/publications/techreports/TR297.html`'.

[Borgida-Mylopoulos-Reiter95]
Alex Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[Chalin02] Back to Basics: Language Support and Semantics of Basic Infinite Integer Types in JML and Larch Technical Report CU-CS 2002-003.2, Computer Science Department, Concordia University, October 2002. Updated March 2003, which is available from the URL
'`http://www.cs.concordia.ca/~chalin/papers/TR-2002-003.pdf`'.

[Chalin04] Patrice Chalin. JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics. *Journal of Object Technology*, 3(6):57-79, June 2004.

[Cheon03] Yoonsik Cheon. A runtime assertion checker for the Java Modeling Language. Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, April 2003. The author's Ph.D. dissertation, which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR03-09/TR.pdf`'.

[Cheon-Leavens02]
Yoonsik Cheon and Gary T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *ECOOP 2002 – Object-Oriented Programming, 16th European Conference, Malaga, Spain*, pages 231–255. Springer-Verlag, June 2002. Also Department of Computer Science, Iowa State University, TR #01-12a, November 2001, revised March 2002 which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR01-12/TR.pdf`'.

[Cheon-Leavens02b]
Yoonsik Cheon and Gary T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). In Hamid R. Arabnia and Youngsong Mun (eds.), *Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA*, pages 322–328. CSREA Press, June 2002. Also Department of Computer Science, Iowa State University, TR #02-05, March 2002 which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR02-05/TR.pdf`'.

[Cheon-Leavens05]
Yoonsik Cheon and Gary T. Leavens. A contextual interpretation of undefinedness for runtime assertion checking. In *AADEBUG 2005, Proceedings of the Sixth International Symposium on Automated and Analysis-Driven Debugging, Monterey, California, September 19–21, 2005*, pages 149–157. ACM Press, September 2005.

[Cohen90] Edward Cohen. *Programming in the 1990s: An Introduction to the Calculation of Programs*. Springer-Verlag, New York, N.Y., 1990.

[Chalin02] Patrice Chalin. Back to Basics: Language Support and Semantics of Basic Infinite Integer Types in JML and Larch. Computer Science Department, Concordia University, Technical Report CU-CS 2002-003.1. Available in '`http://tinyurl.com/e5q6g`', October, 2002.

[Dhara-Leavens96]
Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, pages 258–267. IEEE Computer Society Press, March 1996. An extended version is Department of Computer Science, Iowa State University, TR #95-20b, December 1995, which is available from the URL
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR95-20/TR.ps.Z`'.

[Ernst-etal01]
Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin Dynamically Discovering Likely Program Invariants to Support Program Evolution *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[Finney96] Kate Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[Fitzgerald-Larsen98]
John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development*. Cambridge University Press, Cambridge, UK, 1998.

[Gifford-Lucassen86]
David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on LISP and Functional Programming*, pages 28–38. ACM, August 1986.

[Gosling-Joy-Steele96]
James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, 1996.

[Gosling-etal00]
James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, MA, 2000.

[Gries-Schneider95]
David Gries and Fred B. Schneider. Avoiding the Undefined by Underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 366–373. Springer-Verlag, New York, N.Y., 1995.

[Guttag-Horning93]
John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, New York, N.Y., 1993.

[Hayes93] I. Hayes, editor. *Specification Case Studies*. International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.

[Hoare69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoare72a]
C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Huisman01]
Marieke Huisman. Reasoning about JAVA programs in higher order logic with PVS and Isabelle. IPA dissertation series, 2001-03. Ph.D. dissertation, University of Nijmegen, 2001.

[ISO96] International Standards Organization. *Information Technology - Programming Languages, Their Environments and System Software Interfaces - Vienna Development Method - Specification Language - Part 1: Base language*. International Standard ISO/IEC 13817-1, December, 1996.

[Jacobs-etal98]
Bart Jacobs, Joachim van den Berg, Marieke Huisman Martijn van Berkum, Ulrich Hensel, and Hendrik Tews. Reasoning about Java Classes (Preliminary Report). In *OOPSLA '98 Conference Proceedings*, volume 33, number 10 of *ACM SIGPLAN Notices*, pages 329–340. October 1998.

[Jones90] Cliff B. Jones. *Systematic Software Development Using VDM*. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[Jonkers91]
H. B. M. Jonkers. Upgrading the pre- and postcondition technique. In S. Prehn and W. J. Toetenel, editors, *VDM '91 Formal Software Development Methods 4th International Symposium of VDM Europe Noordwijkerhout, The Netherlands, Volume 1: Conference Contributions*, volume 551 of *Lecture Notes in Computer Science*, pages 428–456. Springer-Verlag, New York, N.Y., October 1991.

[Lano-Haughton94]
K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies*. The Object-Oriented Series. Prentice Hall, New York, N.Y., 1994.

[Leavens96b]
Gary T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In Haim Kilov and William Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, Boston, 1996. An extended version is TR #96-01d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011.

[Leavens97c]
Gary T. Leavens. *Larch/C++ Reference Manual*. Version 5.14. Available in '`ftp://ftp.cs.iastate.edu/pub/larchc++/lcpp.ps.gz`' or on the World Wide Web at the URL
'`http://www.cs.iastate.edu/~leavens/larchc++.html`', October 1997.

[LeavensLarchFAQ]
Gary T. Leavens. Larch frequently asked questions. Version 1.110. Available in '`http://www.cs.iastate.edu/~leavens/larch-faq.html`', May 2000.

[Leavens-etal-JMLRef]
Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, and Joseph Kiniry. *JML Reference Manual*. Available from '`http://www.jmlspecs.org/`'.

[Leavens-Baker99]
Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods: World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999, Proceedings*, volume 1709 of *Lecture Notes in Computer Science*, pages 1087–1106. Springer-Verlag, 1999.

[Leavens-Baker-Ruby99]
Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: a Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds (editors), *Behavioral Specifications for Businesses and Systems*, chapter 12, pages 175-188.

[Leavens-Weihl95]
Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Informatica*, 32(8):705–778, November 1995.

[Leavens-Wing97a]
Gary T. Leavens and Jeannette M. Wing. Protective interface specifications. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development, 7th International Joint Conference CAAP/FASE, Lille, France*, volume 1214 of *Lecture Notes in Computer Science*, pages 520–534. Springer-Verlag, New York, N.Y., 1997.

[Ledgard80]
Henry. F. Ledgard. A human engineered variant of BNF. *ACM SIGPLAN Notices*, 15(10):57–62, October 1980.

[Leino95a] K. Rustan M. Leino. A myth in the modular specification of programs. Technical Report KRML 63, Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue Palo Alto, CA 94301, November 1995. Obtain from the author, at leino@microsoft.com.

[Leino95] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Leino98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA '98 Conference Proceedings*, pp. 144-153. Volume 33, number 10 of *ACM SIGPLAN Notices*, October, 1998.

[Leino-Nelson-Saxe00]
K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java User's Manual. Compaq SRC Technical Note 2000-02, October, 2000.

[Leino-Poetzsch-Heffter-Zhou02]
Using Data Groups to Specify and Check Side Effects. *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pp. 246-257. Volume 37, number 5 of *ACM SIGPLAN Notices*, June, 2002.

[Leino-etal00]
K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking. Web page at '`http://research.compaq.com/SRC/esc/Esc.html`'.

[Lerner91] Richard Allen Lerner. Specifying objects of concurrent systems. Ph.D. Thesis CMU-CS-91-131, School of Computer Science, Carnegie Mellon University, May 1991.

[Liskov-Wing94]
Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Lucassen87]
John M. Lucassen. Types and effects: Towards the integration of functional and imperative programming. Technical Report TR-408, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1987.

[Lucassen-Gifford88]
John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 47–57. ACM, January 1988.

[Luckham-vonHenke85]
David Luckham and Friedrich W. von Henke. An overview of Anna - a specification language for Ada. *IEEE Software*, 2(2):9–23, March 1985.

[Luckham-etal87]
David Luckham, Friedrich W. von Henke, Bernd Krieg-Brückner, and Olaf Owe. *ANNA - A Language for Annotating Ada Programs*, volume 260 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, N.Y., 1987.

[Meyer92a]
Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Meyer92b]
Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[Meyer97] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, N.Y., second edition, 1997.

[Morgan94]
Carroll Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[Morgan-Vickers94]
Carroll Morgan and Trevor Vickers, editors. *On the refinement calculus*. Formal approaches of computing and information technology series. Springer-Verlag, New York, N.Y., 1994.

[Morris87] Joseph M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, December 1987.

[Mueller02]
Peter Müller. Modular Specification and Verification of Object-Oriented Programs. Volume 2262 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.

[Nielson-Nielson-Amtoft97]
H. R. Nielson, F. Nielson, and T. Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In M. Dam, editor, *Proceedings of the Fifth LOMAPS Workshop*, number 1192 in *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Ogden-etal94]
William F. Ogden, Murali Sitaraman, Bruce W. Weide, and Stuart H. Zweben. Part I: The RESOLVE framework and discipline — a research synopsis. *ACM SIGSOFT Software Engineering Notes*, 19(4):23–28, Oct 1994.

[Owre-etal95]
Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[Poll-Jacobs00]
E. Poll and B.P.F. Jacobs. A Logic for the Java Modeling Language JML. Computing Science Institute Nijmegen, Technical Report CSI-R0018. Catholic University of Nijmegen, Toernooiveld 1, 6525 ED Nijmegen, November 2000.

[Poetzsch-Heffter97]
Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[Raghavan-Leavens05]
Arun D. Raghavan and Gary T. Leavens. Desugaring JML Method Specifications. Technical Report 00-03e, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, revised May 2005. Available in `ftp://ftp.cs.iastate.edu/pub/techreports/TR00-03/TR.ps.gz'.

[Rosenblum95]
David S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[Ruby-Leavens00]
Clyde Ruby and Gary T. Leavens. Safely Creating Correct Subclasses without Seeing Superclass Code. In *OOPSLA 2000 Conference on Object-Oriented*

Programming, Systems, Languages, and Applications, Minneapolis, Minnesota, pp. 208-228. Volume 35, number 10 of *ACM SIGPLAN Notices*, October, 2000. Also technical report 00-05d, Department of Computer Science, Iowa State University, Ames, Iowa, 50011. April 2000, revised April, June, July 2000. Available in
'`ftp://ftp.cs.iastate.edu/pub/techreports/TR00-05/TR.ps.gz`'.

[Salcianu-Rinard05]
Alexandru Salcianu and Martin Rinard. Purity and Side Effect Analysis for Java Programs. In Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation. Paris, France January 2005. Available in
'`http://www.mit.edu/~salcianu/publications/vmcai05-purity.pdf`'

[Sivaprasad95]
Gowri Sivaprasad. Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces. Technical Report 95-27a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1995.

[Spivey92] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, New York, N.Y., second edition, 1992.

[Talpin-Jouvelot94]
Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[Tan94] Yang Meng Tan. Interface language for supporting programming styles. *ACM SIGPLAN Notices*, 29(8):74–83, August 1994. Proceedings of the Workshop on Interface Definition Languages.

[Tan95] Yang Meng Tan. *Formal Specification Techniques for Engineering Modular C Programs*, volume 1 of *Kluwer International Series in Software Engineering*. Kluwer Academic Publishers, Boston, 1995.

[Wahls-Leavens-Baker00]
Tim Wahls, Gary T. Leavens, and Albert L. Baker. Executing Formal Specifications with Concurrent Constraint Programming. *Automated Software Engineering*, 7(4):315-343, December, 2000.

[Watt91] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International Series in Computer Science. Prentice-Hall, New York, N.Y., 1991.

[Whitehead-Russell25]
A. N. Whitehead and B. Russell. *Principia Mathematica*. Cambridge University Press, London, second edition. edition, 1925.

[Wills94] Alan Wills. Refinement in Fresco. In Lano and Houghton [Lano-Haughton94], chapter 9, pages 184–201.

[Wing87] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[Wing90a] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.

[Wing83] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Woodcock-Davies96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, International Series in Computer Science, 1996.

[Wright92] Andrew K. Wright. Typing references by effect inference. In Bernd Krieg-Bruckner, editor, *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, volume 582 of *Lecture Notes in Computer Science*, pages 473–491. Springer-Verlag, New York, N.Y., 1992.

Example Index

A

B

D

I

J

M

N

P

U

Concept Index

|

A

B

C

I

J

K

L

M

N

O

P

T

U

V

W

Z