Preliminary Design of JML: A Behavioral Interface Specification Language for Java

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

TR #98-06
June 1998

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

1996 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, theory, Larch, Eiffel, JML; 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, logics of programs, pre- and post-conditions, specification techniques.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

Preliminary Design of JML: A Behavioral Interface Specification Language for Java

Gary T. Leavens[*], Albert L. Baker, and Clyde Ruby
Department of Computer Science, 226 Atanasoff Hall
Iowa State University, Ames, Iowa 50011-1040 USA
leavens@cs.iastate.edu, baker@cs.istate.edu, ruby@cs.iastate.edu

June 4, 1998

Abstract

JML is a behavioral interface specification language tailored to Java. JML is designed to be used by working software engineers, and requires only modest mathematical training. To achieve this goal, JML uses Eiffel-style assertion syntax combined with model-based approach to specifications. However, JML supports quantifiers, specification-only variables, frame conditions, and other enhancements that make it more expressive for specification than Eiffel.

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

1 Introduction

JML stands for "Java Modeling Language." JML is a *behavioral interface specification language* (BISL) [Win87] designed to specify Java [AG98, GJS96] modules. Java *modules* are classes and interfaces.

The main goal of the research presented in this paper is to better understand how to make BISLs (and BISL tools) that are practical and effective for Rockwell and similar 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 an outline of the rest of the paper.

1.1 Behavioral Interface Specification

As a BISL tailored to the specification of Java modules, JML describes two important aspects of a Java module:

- its *interface*, which consists of names and the static information found in Java declarations, and

[*]The work of Leavens and Ruby is supported in part by a grant from Rockwell International Corporation and by NSF grant CCR-9503168. The work of Leavens and Baker is supported in part by the NSF grant CCR 9803843.

```
public class IntMathOps {                                      // 1
  public static int isqrt(int y)                               // 2
       //@ behavior {                                          // 3
       //@   requires y >= 0;                                  // 4
       //@   ensures result * result <= y                      // 5
       //@           && y < (result + 1) * (result + 1);       // 6
       //@ }                                                   // 7
  { return (int) java.lang.Math.sqrt(y); }                     // 8
}                                                              // 9
```

Figure 1: A JML specification written as annotations in the Java code file `IntMathOps.java`.

- its *behavior*, which tells how the module acts when used.

Because they describe interface details for clients written in a specific programming language, BISLs are inherently language-specific [Win87]. For example, a BISL tailored to C++, such as Larch/C++ [Lea97], would describe 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 functions that are specified as native code).

Specifications written in JML are designed to be annotations in Java code files [Tan94, Tan95, LvHKBO87]. To a Java compiler, such annotations are comments that are ignored, This allows JML specifications, such as the specification in Figure 1, to be embedded in Java code files. It is possible, however, to have specifications that are separate from code, if desired; this can be done as in Figure 2. Most of our examples, however, will be Java code files, as we expect most users to use this form. Whatever users prefer, they would only use one of these formats, not both.

As a simple example of a behavioral interface specification in JML, consider the specification in Figure 1. This figure specifies a Java class, `IntMathOps` that contains one static method (function member) named `isqrt`. The comments to the far right give the line numbers in this specification. Comments with an immediately following at-sign (`@`), as on lines 3–7, are *annotations*, which are treated as comments by a Java compiler, but the text following the annotation marker is meaningful in JML.

In Figure 1, interface information is declared in lines 1 and 2. Line 1 declares a class named `ISqrt`, and line 2 declares a method named `isqrt`. Note that all of Java's declaration syntax is allowed in JML, including, on lines 1 and 2, that the names declared are `public`, that the method is `static` (line 2), that its return type is `int` (line 2), and that it takes one `int` argument.

Such interface declarations must be found in a Java module that correctly implements this specification. This automatically the case in Figure 1, 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.

In Figure 1, the behavioral information is specified in the annotations on lines 3–7. The behavioral part of a specification is found between an opening `behavior {` (line 3) and a closing `}` (line 7), ignoring the annotation markers (`//@`). The keyword `behavior` is used

```
public class IntMathOps {
  public static int isqrt(int y);
    behavior {
      requires y >= 0;
      ensures result * result <= y
              && y < (result + 1) * (result + 1);
  }
}
```

Figure 2: A form of the previous specification that lives in a `.jml` file, `IntMathOps.jml`. This form allows the code for concrete methods to be omitted. Note that the semicolon comes before the `behavior` keyword when the code is omitted.

to make the following specification distinct (in the syntax) from the code block that follows it. Between these is a precondition, which follows the keyword `requires` on line 4, and a postcondition, which follows the keyword `ensures` on line 5. 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 in a state that satisfies the postcondition. This is a contract between the caller of the method and the implementor [Hoa69, Mey92a]. 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`. Note that the behavioral specification does not give an algorithm for finding the square root.

As shown in Figure 1, JML can add annotations directly to classes containing Java code. But one can also use JML to write documentation in separate non-Java "`.jml`" files. Since these files are not Java code files, JML allows the user to omit the code for concrete methods in a class. Figure 2 shows how this is done, replacing the code by a semicolon (`;`), as in a Java abstract method declaration.

To summarize, a behavioral interface specification specifies 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 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 the JML specification language will be judged by how well it can document reusable class libraries, frameworks, and Application Programmer Interfaces (APIs) of the kind found at Rockwell.

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 or certain kinds of software, then some APIs would not be amenable to documentation 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 reusable software components. This is especially true since software that is implemented and debugged is more likely to be reused than software that has yet to be implemented.

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

  A preliminary study by Finney [Fin96] indicates that graphic mathematical notations, such as those found in Z [Hay93, Spi92] 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.

  Although we are not promising a formal semantics as part of the research at this stage, we are aiming for it. This aim 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 rang goal of a specification compiler, that would produce prototypes from specifications, if they are constructive.

As a general strategy for achieving these goals, we have tried to blend the Eiffel [Mey92a, Mey92b, Mey97] and Larch [Win87, Win90, GHG$^+$93, Lea98] 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 use the `old` notation from Eiffel, as described below, 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 VDM [Jon90]. 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. The most important is the concept of specification-only declarations. These declarations, as will be explained below, allow more exact specifications of behavior than is typically done in Eiffel. 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 take pains to ban side-effects and other problematic features of code in assertions.

Our experience with Larch/C++, however, has taught us to adapt the model-based approach in several ways, with the aim of making it more practical and easy to learn. An important adaptation is again the use of specification-only model (or ghost) variables. This simplifies the use of JML, as compared with Larch/C++, since users 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 technical advantages that will be described below.)

In the Larch approach to behavioral interface specification [Win87], the mathematical notation used in assertions is presented directly to the user. 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, where the details of the mathematical modeling are hidden behind a facade of Java classes. These classes have to be pure, in the sense that they reflect the underlying mathematics, and hence do not use side-effects (at least not in any observable way). 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.

1.3 Outline

In the next sections we describe more about JML and its semantics. Section 2 uses examples to show how Java classes and interfaces are specified; this section also briefly describes the semantics of subtyping and refinement. Section 3 describes more detail about the expressions that can be used in predicates. Section 4 gives a brief overview of our plans for concurrency and atomicity specification.

2 Class and Interface Specifications

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

2.1 Abstract Models

A simple example of an abstract class specification is the ever-popular `UnboundedStack` type, which is presented in Figure 3. This figure has the abstract values of stack objects specified by the model instance variable `theStack`, which is declared on the third non-blank line. A model variable is a specification-only variable (or ghost variable). It does not have to be implemented, but for purposes of the specification we treat it exactly as any other Java variable. That is, we imagine that each instance of the class `UnboundedStack` has such a variable.

The type of the model variable `theStack` is a JML built-in type, `JMLObjectSequence`, which is a sequence of objects. It is found in the package `edu.iastate.cs.jml.models`, which is imported in the first line of the figure.[1] Note that this `import` declaration does not have to appear in the implementation, since it is modified by the keyword `model`. In general, any declaration form in Java can have this modifier, with the same meaning: that the declaration in question is only used for specification purposes, and does not have to appear in an implementation.

[1] Users can also define their own model types, as we will show below.

```
//@ model import edu.cs.iastate.jml.models.*;

public abstract class UnboundedStack {

  //@ public model JMLObjectSequence theStack;

  //@ public initially theStack.isEmpty();

  public abstract void pop( );
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   modifiable theStack;
    //@   ensures theStack.equals(old theStack.trailer());
    //@ }

  public abstract void push(Object x);
    //@ behavior {
    //@   modifiable theStack;
    //@   ensures theStack.equals(old theStack.addFirst(x));
    //@ }

  public abstract Object top( );
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   ensures result == theStack.first();
    //@ }
}
```

Figure 3: A specification of the abstract class `UnboundedStack` (file `UnboundedStack.java`).

Following the declaration of the model variable, above the specification of pop in Figure 3, is an initially clause. (Such clauses are adapted from Resolve [OSWZ94].) This clause is declared public, since it refers to a public model variable.

An initially clause permits data type induction ([Hoa72, Win83]) for abstract classes and interfaces, by supplying a property that must appear to be true of the starting states of objects. In each visible state (outside of the methods of UnboundedStack) all reachable objects of the type UnboundedStack must have a value that makes them appear to have been created as empty stacks, and subsequently modified using the type's methods.

Following the initially clauses are the expected specifications of the pop, push, and top methods.

The use of the modifiable clauses in the behavioral specifications of pop and push is interesting (and another difference from Eiffel). These give frame conditions [BMR95], which say that no objects, other than those mentioned (and those on which these objects depend, as explained below) may have their values changed.[2] When the modifiable clause is omitted, as it is in the specification of top, this means that no objects can have their state modified by the method's execution. Our interpretation of this is very strict, even benevolent side effects are disallowed if the modifiable clause is omitted [Lei95b, Lei95a].

When a method can modify some objects, these objects have different values in the pre-state and post-state of that method. Often the post-condition must refer to both of them. This is where Eiffel's old notation is used, to refer to the pre-state value of a variable. For example, in the specification of pop, old theStack is the value of the instance variable theStack in the *pre-state* of the method (just after the method is called and parameters have been passed, but before execution of the body). Following Eiffel, the old operator has very high precedence, so that old theStack.trailer() parses the same as (old theStack).trailer().

Note also that, since JMLObjectSequence is an type whose objects are immutable, one is required to use equals instead of == to compare them for equality of values. (Using == would be a mistake, since it would only compare them for object identity, which in combination with new would always yield false.)

The specification of push does not have a requires clause. This means that the method imposes no obligations on the caller. (Logically, 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, following Poetzsch-Heffter [PH97] by releasing implementations from their obligations to fulfill the postcondition when Java runs out of storage. That is, a method implementation is correct if, whenever it is called in a state that satisfies its precondition, either

- the method terminates in a state that satisfies its postcondition, having modified only the objects permitted by its modifiable clause, or
- Java signals an error, by throwing an exception that inherits from Error.

[2]An object is modified by a method when it is allocated in both the pre- and post-states of the method, and when some of its variables (model or concrete) change their values. This means that allocating objects, using Java's new operator, does not cause a modification.

2.2 Dependencies, Representations, and Exceptions

In this subsection we describe how model variables can be related to one another, and how dependencies among them affects the meaning of the `modifiable` clause. For this purpose we give two specifications, `BoundedThing` and `BoundedStack`. Along the way we also demonstrate how to specify methods that can throw exceptions, and other features of JML.

Figure 4 is an interface specification with a simple abstract model. In this case, there are two model variables `MAX_SIZE` and `size`. The variable `MAX_SIZE` is a static field (a class variable), while `size` is thought of as a property of instances (i.e., it is a model instance variable or data member).[3]

In specifications of interfaces that extend or classes that implement this interface, these specification-only variables are inherited. Thus, for example, every object that has a type that is a subtype of the `BoundedThing` interface is thought of, abstractly, as having a field `size`, of type `int`.

Two pieces of class-level specification come after the abstract model in Figure 4. The first is an `invariant` clause.

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 *visible state*; i.e., for each state outside of a method, and at the beginning and end of each method. The figure's invariant says that in every visible state, the `MAX_SIZE` variable has to be positive, and that every reachable object that is a `BoundedThing` must have a size field that has a value less than or equal to `MAX_SIZE`.

Following the invariant is a history constraint [LW94]. A history constraint is used to say how values can change between earlier and later states, such as a method's pre-state and its post-state. This prohibits subtypes from making certain state changes, even if they implement more methods than are specified in a given class. The history constraint in Figure 4 says that the value of `MAX_SIZE` cannot change, since in every pre-state and post-state (before and after the invocation of a method), its value in the post-state, written `MAX_SIZE`, must equal its value in the pre-state, written `old MAX_SIZE`.

Following the history constraint are the interfaces and specifications for four public methods.

The specification of the last method, `clone`, is somewhat interesting. Since `clone` may throw an exception, we use logical implication (written `=>`) and the JML primitive `returns` to say that, when the method returns without throwing an exception, then the result will be a `BoundedThing` and its size will be the same as the model variable `size`. Note the use of the cast in the postcondition of `clone`, which 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 [LW97] since if it is false the meaning of the cast does not matter.

Finally, note that the use of `==` in this Figure 4 is okay, since in each case, the things being compared are primitive values, not references.

Figure 5 gives an interface for bounded stacks that extends the interface in Figure 4. In this specification, one can refer to `MAX_SIZE` from the `BoundedThing` interface, and to `size` as an inherited specification-only field of objects. (We assume that `BoundedThing` is in the same unnamed package for purposes of this paper.)

[3]Java does not allow instance variables to be declared in interfaces, but JML allows model instance variables in interfaces, since these are essential for defining the abstract values of the objects being specified.

```
public interface BoundedThing extends Cloneable {

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

  //@ public invariant MAX_SIZE > 0 && 0 <= size && size <= MAX_SIZE;

  //@ public constraint MAX_SIZE == old MAX_SIZE;

  public int getSizeLimit();
    //@ behavior {
    //@   ensures result == MAX_SIZE;
    //@ }

  public boolean isEmpty( );
    //@ behavior {
    //@   ensures result == (size == 0);
    //@ }

  public boolean isFull();
    //@ behavior {
    //@   ensures result == (size == MAX_SIZE);
    //@ }

  public Object clone () throws CloneNotSupportedException;
    //@ behavior {
    //@   ensures returns => result instanceof BoundedThing
    //@                      && size == ((BoundedThing)result.size);
    //@ }
}
```

Figure 4: A specification of an interface to bounded collection objects (file `BoundedThing.java`).

```
//@ model import edu.cs.iastate.jml.models.*;
public interface BoundedStackInterface extends BoundedThing {
  //@ public model JMLObjectSequence theStack;

  //@ public depends size on theStack;
  //@ public represents size by size == theStack.length();
  //@ public invariant redundantly theStack.length() <= MAX_SIZE;

  //@ public initially theStack.isEmpty();
  //@ public initially redundantly theStack.equals(new JMLObjectSequence());

  public void pop( ) throws BoundedStackException;
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   modifiable size, theStack;
    //@   ensures returns && theStack.equals(old theStack.trailer());
    //@   ensures redundantly theStack.length() == old theStack.length() - 1;
    //@  also
    //@   requires theStack.isEmpty();
    //@   ensures throws(BoundedStackException);
    //@ }

  public void push(Object x ) throws BoundedStackException;
    //@ behavior {
    //@   requires theStack.length() < MAX_SIZE;
    //@   modifiable size, theStack;
    //@   ensures returns && theStack.equals(old theStack.addFirst(x));
    //@   ensures redundantly theStack.length() == old theStack.length() + 1;
    //@  also
    //@   requires theStack.length() >= MAX_SIZE;
    //@   ensures throws(BoundedStackException);
    //@ }

  public Object top( ) throws BoundedStackException;
    //@ behavior {
    //@   requires !theStack.isEmpty();
    //@   ensures returns && result == theStack.first();
    //@  also
    //@   requires theStack.isEmpty();
    //@   ensures throws(BoundedStackException);
    //@ }
}
```

Figure 5: A specification of an interface to bounded stacks (file `BoundedStackInterface.java`).

The abstract model for `BoundedStackInterface` adds to the inherited model by declaring a specification-only field named `theStack`. This field is typed as a `JMLObjectSequence`, as before.

The `depends` and `represents` clauses that follow the declaration of `theStack` are an important feature in modeling with layers of model variables. The `depends` clause says that `size` might change its value when the `theStack` changes, and the `represents` clause says how they are related. The `represents` clause gives additional facts that can be used in reasoning about the specification; in essence it tells how to extract the value of `size` from the value of `theStack`.[4] It serves the same purpose as an abstraction function in various proof methods for abstract data types (such as [Hoa72]).

The `invariant` that follows the `represents` clause in Figure 5 is our first example of checkable redundancy in a specification [LB97, Tan94, Tan95]. This concept is signaled in JML by the use of the keyword `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 invariant follows from 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, since they are then brought 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 checks could be done manually, during reviews, or by an automated tool such as a theorem prover.

Following the invariant, above the specification of `pop` in Figure 5, are two `initially` clauses. The second of these is interesting, in that, since `JMLObjectSequence` is a reference type, one is required to use `equals` instead of `==` to compare them for equality of values. (Using `==` would be a mistake, since it would only compare them for object identity, which in combination with `new` would always yield false.)

Following the `initially` clauses are the specifications of the `pop`, `push`, and `top` methods. These are interesting for several new features that they present. Each of these has its behavioral specification written using two specification cases, separated by the keyword `also`. The semantics is that, when the precondition of a case is satisfied, the rest of that case's specification must be obeyed. In these three cases, the case analysis is only used to separate the specification of the normal case (the first of the two in each method's specification) from the case where an exception in thrown. In the normal case, `returns` is true, whereas when an exception is thrown `throws(BoundedStackException)` is true.

A specification with several cases is shorthand for one in which the cases are combined [DL97, Lea97, Win83, Wil94]. In Figure 6 we show the expanded specification of `pop` from Figure 5. As can be seen from this example, the expanded specification has a postcondition that is a conjunction of implications, one for each case. The impliciation for a case in the expanded postcondition says that when the precondition for that case holds, the case's postcondition must also hold.. The `modifiable` clause for the expanded specification is the union of all the modifiable clauses for the cases; because of this the variables that are named in the combined modifiable clause but not allowed to be modified in a particular case have to be asserted to be unmodified in that case. In this expansion, the model variables `size` and `theStack` are asserted to be unmodified in the second case's translation.

[4]Of course, one could specify `BoundedStack` without separating out the interface for `BoundedThing`, and in that case, this abstraction would be unnecessary. We have made this separation partly to demonstrate more advanced features of JML, and partly to fit the figures on single pages.

```
public void pop( ) throws BoundedStackException;
  //@ behavior {
  //@   requires !theStack.isEmpty() || theStack.isEmpty();
  //@   modifiable size, theStack;
  //@   ensures (!theStack.isEmpty() =>
  //@              returns && theStack.equals(old theStack.trailer()))
  //@        && (theStack.isEmpty() =>
  //@                   throws(BoundedStackException)
  //@                   && unmodified(size, theStack);
  //@   ensures redundantly !theStack.isEmpty() =>
  //@              theStack.length() == old theStack.length() - 1;
  //@ }
```

Figure 6: An expansion of `pop`'s specification. The precondition reduces to `true`, but the precondition shown is the general form for the expansion.

The `depends` clause is important in "loosening up" the `modifiable` clause, for example to permit the fields of an object that implement the abstract model to be changed [Lei95b, Lei95a]. This "loosening up" also applies to model variables that have dependencies declared. For example, since `size` depends on `theStack`, i.e., `size` is in some sense represented by `theStack`, if `size` is mentioned in a `modifiable` clause, then `theStack` is implicitly allowed to be modified. However, just mentioning `theStack` would not permit `size` to be modified, because `theStack` does not depend on `size`; hence for rhetorical purposes, we mention both `size` and `theStack` in the modifiable clauses of `pop` and `push`.

Finally, there is more redundancy in the specifications of `pop` and `push`, which each have a redundant `ensures` clause in their normal case. For a redundant `ensures` clause, what one checks is that the conjunction of the precondition, the meaning of the `modifiable` clause, and the (non-redundant) postcondition itself imply the redundant postcondition. It is interesting to note that the specifications for stacks written in Eiffel [Mey97, page 339] expresses not much more than what we specify in the redundant postconditions of `pop` and `push`. These convey strictly less information than the non-redundant postconditions, since they say little about the elements of the stack.[5]

2.3 Making New Abstract Model Types

JML comes with a suite of built-in modeling types, including `JMLObjectSet`, `JMLObjectSequence`, and `JMLValueSet` and sequences. Users can also create their own modeling types if desired. Since these types are to be treated as purely immutable values in specifications, they must pass certain conservative checks that make sure there is no possibility of observable side-effects from using such objects.

[5]Meyer's specification actually says what the top element of the stack is after a `push`, but says nothing about the rest of the elements. Meyer's second specification and implementation of stacks [Mey97, page 349] is no better in this respect, although, of course, the implementation does keep track of the elements properly.

The extension mechanism uses the modifier `pure`, as in Figure 7. A correct implementation of a `pure` class or interface must be such that:

- all methods must be specified to modify nothing,
- all concrete fields (data members) can only be assigned once, in the constructor, and must be of some primitive value type or a pure type,
- all methods called in the implementation must be declared to be pure (in a specification, either for the method or their class must be pure), and
- a pure class only extends other pure classes.

A pure method can be declared in any class. Such a method must have a specification that nothing is modifiable (in all cases), and a correct implementation can only call other pure methods.

Recursion is permitted, both in pure methods and in data members of pure classes. However, pure methods must be proved to terminate on all inputs.

As an example, we specify a new type, `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 in Figure 7. The abstract model here is a single field of the primitive Java type `long`, which holds a number of pennies.

This interface has a history constraint, which says that the number of pennies in an object cannot change.[6]

The interesting aspect of the operations is another kind of redundancy, given by the `example` clauses [Lea96, LB97]. Any number of examples can be given for a specification case. Here there are two examples in the specification of `dollars` and one in the specification of `cents`. An example's predicate should, when conjoined with any precondition and the modifiable clause should imply the post-condition given. (Note that this is the opposite direction of implication from a redundant ensures clause.) Typically, examples are concrete, and serve to point out various rhetorical points about the use of the specification to the reader. (Exercise: check that all the examples given are correct!)

The interface `Money` is specified to extend the interface `JMLType`. This interface is given in Figure 8. It says that objects should have `equals` and `clone` methods.

The specification of `JMLType` is noteworthy in its use of informal predicates [Lea96]. In this instance, the informal predicates are used as an escape from formality. The use of informal predicates avoids the delicate issues of saying what observable aliasing means, and what equality of values means.

As specified in Figure 7, the type `Money` lacks some useful operations. The extensions in Figures 9 and 10 provide specifications of comparison operations and arithmetic, respectively.

The specification of `MoneyOps` is interesting for the use of a model (specification-only) method, `inRange`. This method cannot be invoked by Java programs. When used in a predicate, `inRange(l)` is equivalent to using some correct implementation of its specification. The specification of `inRange` also makes use of a local model variable declaration.

[6]There is no `initially` clause 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.

```
import edu.iastate.cs.jml.models.JMLType;

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

  //@ public constraint pennies == old pennies;

  public long dollars();
  //@ behavior {
  //@   ensures result == pennies / 100;
  //@   example pennies == 703 && result == 7;
  //@   example pennies == 799 && result == 7;
  //@   example pennies == -503 && result == -5;
  //@ }

  public long cents();
  //@ behavior {
  //@   ensures result == pennies % 100;
  //@   example pennies == 703 && result == 3;
  //@   example pennies == -503 && result == -3;
  //@ }

  public boolean equals(Object o2);
  //@ behavior {
  //@   ensures result == (o2 instanceof Money
  //@                      && pennies == (Money)o2.pennies);
  //@ }

  public Object clone();
  //@ behavior {
  //@   ensures result instanceof Money
  //@           &&(Money)result.pennies == pennies;
  //@ }
}
```

Figure 7: A specification of the pure interface `Money` (file `Money.java`).

```
// @(#)$Id: JMLType.java,v 1.7 1998/06/03 06:39:30 leavens Exp $

package edu.iastate.cs.jml.models;

public interface JMLType extends Cloneable {
  public Object clone();
  /*@ behavior {
    @   ensures result.equals(this)
    @           && informally "result and this are not observably aliased";
    @ }
    @*/
  public boolean equals(Object op2);
  /*@ behavior {
    @   ensures informally "result is true when the value of op2"
    @                      "is not distinguishable from the value of this";
    @ }
    @*/
}
```

Figure 8: A specification of the interface `JMLType` (file `JMLType.java`).

Such declarations allow one to abbreviate long expressions, or, to make rhetorical points by naming constants, as is done with `epsilon`.

2.4 Implementation of Class and Interface Specifications

The key to proofs that an implementation of a class or interface specification is correct lies in the use of `depends` and `represents` clauses [Hoa72, Lei95b]. Consider, for example, the abstract class `MoneyAC` given in Figure 11. It declares a concrete instance variable `numCents`, which is related to the specification-only instance variable `pennies` by the `represents` clause. This allows relatively trivial proofs of the correctness of the `dollars` and `cents` methods, and is key to the proofs of the other methods.

The straightforward implementation of the subclass `MoneyComparableAC` is given in Figure 12. Note that the abstract and concrete variables are both inherited by this class.

An interesting feature of the class `MoneyComparableAC` is the protected method `totalCents`. For this method, we give both a specification and its code. In this case the specification serves as documentation for the code. Note that the specification-only method, `inRange` is not implemented, and does not need to be implemented to make this class correctly implement the interface `MoneyComparable`.

Finally, a concrete class implementation is the class `USMoney`, presented in Figure 13. This class implements the interface `MoneyOps`. Note that specifications as well as code are given for the constructors.

The first constructor's specification illustrates that redundancy can also be used in a `modifiable` clause. In this case the redundant modifiable clause follows from the given modifiable clause and the meaning of the `depends` clause inherited from the superclass

```
public /*@ pure @*/ interface MoneyComparable extends Money
{
  public boolean greaterThan(Money m2);
  //@ behavior {
  //@   requires m2 != null;
  //@   ensures result == (pennies > m2.pennies);
  //@ }

  public boolean greaterThanOrEqualTo(Money m2);
  //@ behavior {
  //@   requires m2 != null;
  //@   ensures result == (pennies >= m2.pennies);
  //@ }

  public boolean lessThan(Money m2);
  //@ behavior {
  //@   requires m2 != null;
  //@   ensures result == (pennies < m2.pennies);
  //@ }

  public boolean lessThanOrEqualTo(Money m2);
  //@ behavior {
  //@   requires m2 != null;
  //@   ensures result == (pennies <= m2.pennies);
  //@ }
}
```

Figure 9: A specification of the pure interface `MoneyComparable` (file `MoneyComparable.java`).

```
public /*@ pure @*/ interface MoneyOps extends MoneyComparable
{
  //@ model public boolean inRange(double d);
  //@ behavior {
  //@   model double epsilon = 1.0;
  //@   ensures result == (Long.MIN_VALUE + epsilon < d
  //@                       && d < Long.MAX_VALUE - epsilon);
  //@ }

  public Money plus(Money m2);
  //@ behavior {
  //@   requires m2 != null && inRange((double) pennies + m2.pennies);
  //@   ensures result != null
  //@           && result.pennies == this.pennies + m2.pennies;
  //@   example this.pennies == 300 && m2.pennies == 400
  //@           && result.pennies == 700;
  //@ }

  public Money minus(Money m2);
  //@ behavior {
  //@   requires m2 != null && inRange((double) pennies - m2.pennies);
  //@   ensures result != null
  //@           && result.pennies == this.pennies - m2.pennies;
  //@   example this.pennies == 400 && m2.pennies == 300
  //@           && result.pennies == 100;
  //@ }

  public Money scaleBy(double factor);
  //@ behavior {
  //@   requires inRange(factor * pennies);
  //@   ensures result != null && result.pennies == (long)(factor * pennies);
  //@   example pennies == 400 && factor == 1.01
  //@           && result.pennies == 404;
  //@ }
}
```

Figure 10: A specification of the pure interface `MoneyOps` (file `MoneyOps.java`).

```
public /*@ pure @*/ abstract class MoneyAC implements Money {

  protected long numCents;
  //@ protected depends pennies on numCents;
  //@ protected represents pennies by pennies == numCents;

  //@ protected constraint redundantly numCents == old numCents;

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

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

  public boolean equals(Object o2)
  {
    try {
      Money m2 = (Money)o2;
      return numCents == (100 * m2.dollars() + m2.cents());
    } catch (ClassCastException e) {
      return false;
    }
  }

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

Figure 11: A pure abstract class `MoneyAC` that implements the interface `Money` (file `MoneyAC.java`).

```
public /*@ pure @*/ abstract class MoneyComparableAC
    extends MoneyAC implements MoneyComparable
{
  protected final long totalCents(Money m2)
       //@ behavior {
       //@   ensures result == m2.pennies;
       //@ }
  {
    return 100 * m2.dollars() + m2.cents();
  }

  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);
  }
}
```

Figure 12: A pure abstract class `MoneyComparableAC` that implements the interface `MoneyComparable` and extends the class `MoneyAC` (file `MoneyComparableAC.java`).

```
public /*@ pure @*/ class USMoney
                extends MoneyComparableAC implements MoneyOps
{
  public USMoney(long cs)
       //@ behavior {
       //@   modifiable pennies;
       //@   modifiable redundantly pennies, numCents;
       //@   ensures pennies == cs;
       //@   ensures redundantly numCents == cs;
       //@ }
  {
    numCents = cs;
  }

  public USMoney(double amt)
       //@ behavior {
       //@   modifiable pennies;
       //@   ensures pennies == (long)(100.0 * amt);
       //@   ensures redundantly informally "pennies represents amt dollars";
       //@ }
  {
    numCents = (long)(100.0 * amt);
  }

  public Money plus(Money m2)
  {
    //@ assert m2 != null;
    return new USMoney(numCents + totalCents(m2));
  }

  public Money minus(Money m2)
  {
    //@ assert m2 != null;
    return new USMoney(numCents - totalCents(m2));
  }

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

Figure 13: A pure concrete class `USMoney` (file `USMoney.java`).

MoneyAC.

The second constructor in Figure 13 is noteworthy in that there is a redundant ensures clauses that uses an informal predicate [Lea96]. In this instance, the informal predicate is used as a comment (which could also be used). Often, however, informal predicates allow an escape from formality when one does not wish to give part of a specification in formal detail.

2.5 Use of Pure Classes

Since USMoney is a pure class, it can be used to make models of other classes. An example is the class BankAccount given in Figure 14. The first model variable in this class has type USMoney.

2.6 Composition for Container Classes

The following example specification of a class Digraph (directed graph) gives a more interesting example of the JML way that more complex models are composed from existing pure classes, model classes, and the intrinsic containers provided in the package edu.iastate.cs.jml.models.

Figure 15 contains an abstract class NodeType. NodeType is an abstract class (rather than a model class) because it will require an implementation and does appear in the interface of model class Digraph. However, we also denote the abstract class as pure, since we will also use NodeType in the specification of other classes. (And we do so appropriately, since all the methods for class NodeType are side-effect-free, as described elsewhere.) In the abstract class specification for NodeType we simply provide a model variable iD, which would represent a unique identifier for nodes. We specify that the equals method for class NodeType simply tests whether two references to objects of type NodeType are references to objects with the same iD's. We also require that NodeType have a public clone method that behaves as we expect such methods to behave.

Figure 16 contains the specification for a model class ArcType. We will use ArcType in the model for Digraph, but ArcType does not require an implementation in that it does not appear in the interface to Digraph. Of course, in declaring ArcType to be a model class we are obligated to insure that all the methods of the class are side-effect-free. The two model variables 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 from and to model variable values. We will require that ArcType support a public clone method as it is required for the container we will use for the type of one of the model variables in Digraph. We will also make use of the constructor in the specification of Digraph.

Finally, the specification of the model class Digraph in Figures 17, 18, and 19 demonstrates how to use container classes to compose models in JML. Both model variables nodes and arcs are of type JMLValueSet. However, in the first invariant clause we restrict 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 we are using JMLValueSet, membership is determined by the use of the equals method for the type of the elements (rather than reference equality).

Thus, in JML, one uses containers like JMLValueSet, combined with appropriate invariants to specify models that are compositions of other existing pure classes, abstract pure

```
public class BankAccount {
  //@ public model USMoney credit;
  //@ public model String owner;
  //@ public invariant credit >= new USMoney(0);
  //@ public constraint owner == old owner;
  public BankAccount(Money amt, String own);
  //@ behavior {
  //@    requires new USMoney(1) <= amt;
  //@    modifiable credit, owner;
  //@    ensures credit.equals(amt) && owner.equals(own);
  //@ }
  public Money balance();
  //@ behavior {
  //@    ensures result.equals(credit);
  //@ }
  public void payInterest(double rate);
  //@ behavior {
  //@    requires 0.0 <= rate && rate <= 1.0;
  //@    modifiable credit;
  //@    ensures credit == old credit.scaleBy(1.0 + rate);
  //@    example rate == 0.05 && old credit.equals(new USMoney(4000))
  //@         && credit.equals(new USMoney(4200));
  //@  }
  public void deposit(Money amt);
  //@ behavior {
  //@    requires amt >= new USMoney(0);
  //@    modifiable credit;
  //@    ensures credit.equals(old credit.plus(amt));
  //@    example old credit.equals(new USMoney(40000))
  //@         && amt.equals(new USMoney(1))
  //@         && credit.equals(new USMoney(40001));
  //@  }
  public void withdraw(Money amt);
  //@ behavior {
  //@    requires 0 <= amt && amt <= old credit;
  //@    modifiable credit;
  //@    ensures credit.equals(old credit.minus(amt));
  //@    example old credit.equals(new USMoney(40001))
  //@         && amt.equals(new USMoney(40000))
  //@         && credit.equals(new USMoney(1));
  //@  }
}
```

Figure 14: Specification of a pure concrete class `BankAccount` (file `BankAccount.java`).

```
import edu.cs.iastate.jml.models.*;

public /*@ pure @*/ abstract class NodeType implements JMLType {

  //@ public model int iD;

  public abstract boolean equals(Object o);
  //@ behavior {
  //@   requires o instanceof NodeType;
  //@   ensures result == (iD == (NodeType)o.iD);
  //@ also
  //@   requires !(o instanceof NodeType);
  //@   ensures result == false;
  //@ }

  public abstract Object clone();
  //@ behavior {
  //@   ensures result instanceof NodeType
  //@     && (NodeType)result.equals(this) && result != this;
  //@ }

}  // end of class NodeType declaration
```

Figure 15: First part of an abstract class specification `NodeType` (file `NodeType.java`).

```
import edu.cs.iastate.jml.models.JMLType;

public model abstract class ArcType implements JMLType {

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

  public ArcType(NodeType inFrom, NodeType inTo);
    behavior {
      requires inFrom != null && inTo != null;
      modifiable from, to;
      ensures from.equals(inFrom) && to.equals(inTo)
              && from != inFrom && to != inTo;
    }

  public model boolean equals(Object o);
    behavior {
      requires o instanceof ArcType;
      ensures result == (    (ArcType)o.from.iD == from.iD
                          && (ArcType)o.to.iD == to.iD );
    also
      requires !(o instanceof ArcType);
      ensures result == false;
    }

  public Object clone();
    behavior {
      ensures result instanceof ArcType
              && (ArcType)result.equals(this) && result != this;
    }
}
```

Figure 16: First part of a model class specification `ArcType` (file `ArcType.jml`).

```
import edu.cs.iastate.jml.models.*;
public class Digraph {

 public model JMLValueSet nodes;
 public model JMLValueSet arcs;

 public invariant forall (Object n) [ nodes.isIn(n) => n instanceof NodeType];
 public invariant forall (Object a) [ arcs.isIn(a) => a instanceof ArcType];
 public invariant forall (ArcType a)
       [ arcs.isIn(a) => nodes.isIn(a.from) && nodes.isIn(a.to) ];

 public Digraph();
   behavior {
     modifiable nodes, arcs;
     ensures nodes.equals(new JMLValueSet())
             && arcs.equals(new JMLValueSet());
   }
```

Figure 17: First part of a class specification `Digraph` (file `Digraph.jml`).

classes, or model classes.

The other features of JML used in the specification of `Digraph` appear elsewhere. One interesting use of model methods (those used solely for writing assertions in specifications and that do not impose an obligation for implementation and that are not part of the interface for a class). The model method `ReachSet` constructively defines the set of all nodes that are reachable from the nodes in the argument `nodeSet`. Note the recursive ensures clause for `ReachSet`, which 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`.

2.7 Subtyping

Following Dhara and Leavens [DL96, Lea97], a subtype inherits the specifications of its supertype's public and protected member functions, as well as invariants and history constraints. This ensures that a subclass specifies a behavioral subtype of its supertypes; This inheritance can be thought of textually, by copying the specifications of the methods of a class's ancestors and all interfaces that a class implements into the class, and treating combinations by adding the old specification in as a new specification case.

For example, consider `PlusAccount` as a subclass of `BankAccount`. It inherits the fields from `BankAccount`, and the initially clauses, invariants, and history constraints from `BankAccount`. Because it inherits the fields of its superclasses, the specifications of those classes are still meaningful when copied to the subclass. The trick is to always add new model variables to the subclass and relate them to the existing ones.

```
public void addNode(NodeType n);
  behavior {
    requires n != null;
    modifiable nodes;
    ensures nodes.equals(old nodes.insert(n));
  }

public void removeNode(NodeType n);
  behavior {
    requires unconnected(n);
    modifiable nodes;
    ensures nodes.equals(old nodes.remove(n));
  }

public void addArc(NodeType inFrom, NodeType inTo);
  behavior {
    requires inFrom != null && inTo != null
             && nodes.isIn(inFrom) && nodes.isIn(inTo);
    modifiable arcs;
    ensures arcs.equals(old arcs.insert(new ArcType(inFrom, inTo)));
  }

public pure boolean isNode(NodeType n);
  behavior {
    ensures result == (nodes.isIn(n));
  }

public pure boolean isArc(NodeType inFrom, NodeType inTo);
  behavior {
    ensures result == (arcs.isIn(new ArcType(inFrom, inTo)));
  }

public pure boolean isAPath(NodeType start, NodeType end);
  behavior {
    requires nodes.isIn(start) && nodes.isIn(end);
    ensures result == ReachSet(new JMLValueSet().insert(start)).isIn(end);
  }
```

Figure 18: Second part of a class specification `Digraph` (file `Digraph.jml`, continued).

```
 public model boolean unconnected(NodeType n);
   behavior {
     ensures result == !exists (ArcType a) [ arcs.isIn(a)
                       && (a.from.equals(n) || a.to.equals(n)) ];
   }

 public model JMLValueSet ReachSet(JMLValueSet nodeSet);
   behavior {
     requires nodeSet != null
       && forall (Object o) [nodeSet.isIn(o) =>
            o instanceof NodeType && nodes.isIn(o)];
     ensures
       (nodeSet.equals(OneMoreStep(nodeSet)) => result .equals(nodeSet))
       && (!nodeSet.equals(OneMoreStep(nodeSet)) =>
             result .equals(ReachSet(OneMoreStep(nodeSet))) );
   }

 public model JMLValueSet OneMoreStep(JMLValueSet nodeSet);
   behavior {
     requires nodeSet != null
       && forall (Object o) [nodeSet.isIn(o) =>
            o instanceof NodeType && nodes.isIn(o)];
     ensures result. equals(nodeSet.union(
       new JMLValueSet { NodeType n |
          exists (ArcType a) [arcs.isIn(a) &&
            (    nodeSet.isIn(a.from) && n.equals(a.to)
              || nodeSet.isIn(a.to) && n.equals(a.from) )]}));
   }
}  // end of class Digraph
```

Figure 19: Third part of a class specification `Digraph` (file `Digraph.jml`, continued).

3 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 `--`), and

We allow the allocation of storage (e.g., using operator `new`) in predicates, because such storage can never be referred to after the evaluation of the predicate.

JML adds the following new syntax to the Java expression syntax, for use in predicates:

- `=>` for logical implication; for example, `raining => getsWet` is true if either `raining` is false or `getsWet` is true.
- `forall` and `exists`, which are quantifiers; 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.
- `returns`, which is true if a method returns normally, when an exception is thrown this is false.
- `result`, which, when `returns` is true, is the object that is the result of the method.
- `throws`, which can be used to assert that a particular exception is thrown; for example `throws(ArithmeticException)` is true when the exception `ArithmeticException` is thrown.
- `thrown`, which can be used to describe the object that is the "exception result" when a method throws an exception; for example when `throws(ArithmeticException)` is true, then `thrown(ArithmeticException)` is the object that was thrown.
- `fresh`, which asserts that objects were freshly allocated; for example, `fresh(x,y)` asserts that the objects bound to `x` and `y` were not allocated in the pre-state.
- `old`, which can be used to refer to objects and their values in the pre-state; for example, `old myPoint.x` is the value of the `x` field of the object `myPoint` in the pre-state.
- `unmodified`, which asserts that the values of objects are the same in the post-state as in the pre-state; for example, `unmodified(xval,yval)` says that `xval` and `yval` have the same value in the pre- and post-states.
- `reach`, which returns a `JMLObjectSet` of all objects reachable from a given object.
- Set comprehensions, which can be used to succinctly define sets; for example, the following is the `JMLObjectSet` of `Integer` objects whose values are between 0 and 10, inclusive.

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

As in Java itself, most types are reference types, and hence many expressions yield references (i.e., object identities or addresses), not 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, except for primitive values, one has to use 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 the values of two strings one must write `myString.equals(yourString)`.

The reference semantics makes interpreting predicates that involve the use of `old` interesting. We want to have the semantics suited for two purposes:

- execution of assertions for purposes of debugging and testing, as in Eiffel, and
- generation of mathematical assertions for static analysis and possible theorem proving (e.g., to verify program correctness).

See Appendix A for details on the mathematical modeling of assertions.

To use predicates to debug and test Java code one has to have a way to refer to old values and objects without making a copy of the entire state, since that would involve all of the computer's memory. This is no problem when primitive values or pure (immutable) objects are used for all variables referred to with `old`, since the value or reference stored in such a variable can simply be saved. One way to deal with mutable objects is to simply not execute assertions involving references to `old` and such objects. However, we hope to make more progress in this area.

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 [GL86, Luc87, LG88, NNA97, TJ94, Wri92] to make sure that only pure methods may be called in predicates.

Exceptions in expressions are particularly important, since they may arise in type casts. Logically, we will deal 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 result type is a reference type, this means that we will 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.

We will check that errors are not explicitly thrown by pure methods. This means that they can be ignored during mathematical modeling. When executing predicates, errors will also be ignored, but will cause run time errors.

4 Concurrency

For concurrency our plan is to use `when` clauses that say when a method may proceed to execute, after it is called [Ler91, Siv95]. This permits the specification of when the caller is delayed to obtain a lock, for example. While syntax for this exists in the JML parser, our exploration of this topic is still in an early stage.

Acknowledgements

Thanks to Rustan Leino and Peter Mueller for many discussions about the semantics of such specifications. For comments on an earlier draft, thanks to Peter, Anand Ganapathy, Sevtap Oltes, Gary Daugherty, Karl Hoech, Jim Potts, and Tammy Scherbring.

References

[AG98] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley, Reading, MA, second edition, 1998.

[BMR95] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, 21(10):785–798, October 1995.

[DL96] 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.

[DL97] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. Technical Report 95-20c, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1997. Also in Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996, pp. 258–267. Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu.

[Fin96] Kate Finney. Mathematical notation in formal specification: Too difficult for the masses? *IEEE Transactions on Software Engineering*, 22(2):158–159, February 1996.

[GHG$^+$93] 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.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* The Java Series. Addison-Wesley, Reading, MA, 1996.

[GL86] 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.

[Hay93] I. Hayes, editor. *Specification Case Studies.* International Series in Computer Science. Prentice-Hall, Inc., second edition, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hoa72] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[Jon90] Cliff B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[LB97] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. Technical Report 97-19, Iowa State University, Department of Computer Science, September 1997.

[Lea96] 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.

[Lea97] 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.

[Lea98] Gary T. Leavens. Larch frequently asked questions. Version 1.89. Available in http://www.cs.iastate.edu/~leavens/larch-faq.html, January 1998.

[Lei95a] 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 rustan@pa.dec.com.

[Lei95b] K. Rustan M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[Ler91] Richard Allen Lerner. Specifying objects of concurrent systems. Ph.D. Thesis CMU-CS-91-131, School of Computer Science, Carnegie Mellon University, May 1991.

[LG88] 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.

[LH94] K. Lano and H. Haughton, editors. *Object-Oriented Specification Case Studies.* The Object-Oriented Series. Prentice Hall, New York, N.Y., 1994.

[Luc87] 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.

[LvHKBO87] 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.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LW97] 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.

[Mey92a] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

[Mey92b] Bertrand Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, New York, N.Y., second edition, 1997.

[NNA97] 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.

[OSWZ94] 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.

[PH97] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* PhD thesis, Technische Universität München, 1997. (Habilitationsschrift).

[Siv95] 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.

[Spi92] J. Michael Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice-Hall, New York, N.Y., second edition, 1992.

[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.

[TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

[Wil94] Alan Wills. Refinement in Fresco. In Lano and Houghton [LH94], chapter 9, pages 184–201.

[Win83] Jeannette Marie Wing. A two-tiered approach to specifying programs. Technical Report TR-299, Massachusetts Institute of Technology, Laboratory for Computer Science, 1983.

[Win87] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–24, September 1990.

[Wri92] 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.

A Details for Mathematical Modeling of Expressions

Mathematically, the semantics of a language like Java can be given in terms of an environment and a state (sometimes called a store). The environment maps identifiers (such as formal parameter names) to references (locations), and the state maps references to values.

$$\begin{aligned} \mathit{Environment} &= \mathit{Identifier} \to \mathit{Reference} && (1) \\ \mathit{State} &= \mathit{Reference} \to \mathit{Value} && (2) \\ \mathit{Value} &= \mathit{PrimitiveValue} + \mathit{Reference} && (3) \end{aligned}$$

The main problems are:

- how mathematical functions can mimic Java methods, which have implicit access to the state, and
- how to make sure that the right state is accessed when `old` is used in assertions.

Our solution to these problems, in the mathematical world, is to make the meaning of an expression be either a primitive value or a pair consisting of a reference and a state. We abuse terminology and call such a pair an *object*.

$$\begin{aligned} \mathit{ExpressibleValue} &= \mathit{PrimitiveValue} + \mathit{Object} && (4) \\ \mathit{Object} &= \mathit{Reference} \times \mathit{State} && (5) \end{aligned}$$

Having the state as part of an object allows mathematical functions to mimic the behavior of Java methods, which can implicitly access the state. It also allows the right state to be used for `old` expressions, since the state paired with the object that is the result of an old expression will be the pre-state of a method, instead of the current state.

(The meaning of expressions used in predicates can be mathematically modeled as a (curried) function that takes an environment and a pair of states and returns an expressible value. The first state in the pair passed to the meaning function is the pre-state for the method, the second is the current state (in the post-condition, this starts out as the post-state). The following shows the type of the meaning function for expressions, and the cases for identifiers and the use of `old`.

```
meaning: $\mathit{Expression} \to \mathit{Environment} \to (\mathit{State} \times \mathit{State}) \to \mathit{ExpressibleValue}$
meaning($I$) $e$ $(pre, curr)$ =
  **let** $v = curr(e(I))$
  **in if** isPrimitiveValue($v$) **then** $v$ **else** $(v, curr)$ **fi**
meaning(old $E$) $e$ $(pre, curr)$ = meaning($E$) $e$ $(pre, pre)$
```

This shows that an `old` expression is evaluated in the pre-state, and if it evaluates to an object, the state of the object will be the pre-state.)