Improving JML: For a Safer and More Effective Language

Patrice Chalin
www.cs.concordia.ca/~faculty/chalin

Technical Report 2003-001.3
June 2003

Faculty of Engineering and Computer Science (ENCS)
Department of Computer Science
Concordia University

Keywords: behavioral interface specification languages, Java Modeling Language, JML, specification language design and semantics, arbitrary precision numeric types, formal methods

2001 CR Categories: D.2.1 [Software Engineering] Requirements/ Specifications — languages, tools, JML; D.2.2 [Software Engineering] Design Tools and Techniques; D.2.4 [Software Engineering] Software/Program Verification; D.3.2 [Programming Languages] Language Classifications — Object-oriented languages; F.3.1 [Logics and Meanings of Programs] Specifying and Verifying and Reasoning about Programs.

Revision History

2003/03/16, revision 1	First public release.
2003/05/06, revision 2	In Section 4.1: • Replaced description of "5.6 Numeric Promotions" by specific descriptions for unary numeric promotion and binary numeric promotion. [04/22]. • Removed the statement that JMLa literals are to be of type integer or real because Java does not support implicit narrowing of integer constants in method invocations. Added a specific rule for '*Conversion from* `real` to `integer`.' Minor changes, e.g., Figure 6: changed '`isRange`' to '`inRange`'.
2003/06, revision 3	• Type names are now `\bigint` and `\real`. • Operators involved implicit promotion are: unary -, binary +, -, *, / and %. • Implicit promotion does not occur for constant expressions, unless the resulting value is out of range. • Discussion of similar issues with Eiffel. • Other minor corrections.

Table of Contents

Improving JML: For a Safer and More Effective Language

Patrice Chalin

Computer Science Department, Concordia University, Canada
www.cs.concordia.ca/~faculty/chalin

Abstract. An unusually high number of published JML specifications are invalid or inconsistent, including cases from the security critical area of smart card applications. We claim that these specification errors are due to a mismatch between user expectations and the current JML semantics of expressions over numeric types. At the heart of the problem is JML's language design decision to assign to arithmetic operators the same semantics as in Java. Consequently, JML arithmetic is bounded in precision and more importantly loss of precision occurs stealthily. After a short discussion of JML language design goals and objectives, we introduce JMLa, an adaptation of JML supporting *primitive arbitrary precision* numeric types. To support our claim that the identified specification errors are due to JML's divergence from user expectations, we demonstrate that the invalidities and inconsistencies disappear under JMLa semantics with either no, or minor syntactic changes to the specifications. Other advantages of JMLa are illustrated including safety—how it allows an automated static checker like ESC/Java to detect more specification and implementation errors. We also briefly illustrate how these issues are applicable to other assertion-based languages like Eiffel.

Keywords: behavioral interface specification languages, Java Modeling Language, JML, specification language design and semantics, arbitrary precision numeric types, assertion-based languages, formal methods.

1 Introduction

1.1 JML

The Java Modeling Language, JML, is a notation for specifying and describing the detailed design and implementation of Java modules. It is a model-based specification language offering, in particular, method specification by pre- and post-condition and class invariants to document required module behavior. JML is an open collaborative project. Gary Leavens of Iowa State University is coordinating JML language design efforts as well as the development of tools such as a JML type checker and run-time assertion-checker compiler. An Extended Static Checker, ESC/Java, developed at the Compaq Systems Research Center, uses a subset of JML to automatically carry out the (partial) verification of Java code based on its JML specifications. Also, the complete formal verification of Java modules can be achieved using the LOOP tool, from the University of Nijmegen. The LOOP tool can be used to automatically translate Java modules and their corresponding JML specifications into the language of the Prototype Verification System (PVS) [Owre96], an interactive theorem prover. PVS is then used to carry out the correctness proofs [Leavens+00]. Other tools that process JML specifications are described at JMLspecs.org.

JML is the product of over two decades of research in the area of Behavioral Interface Specification Language (BISL) design. The main goal driving its evolution has been to explore ways of creating a BISL that is both *practical* and *effective* [LBR02]. This paper focuses on an issue that is fundamental to most design specification languages: support for arbitrary precision integers and reals. This is in contrast to the programming language *approximations* to these numeric types. At the heart of the problem addressed in this paper is the JML design decision to assign to arithmetic operators the same semantics as in Java. Consequently, JML arithmetic is bounded in precision and more importantly, loss of precision occurs stealthily. As will be explained in Section 2, JML does offer support for arbitrary precision integers by means of the `JMLInfiniteInteger` model class, but use of such a class results in verbose specifications that rapidly become unclear and difficult to understand.

The main contributions of this paper are:

- A proposed refinement of the JML language design goals into specific objectives (Section 3).
- A proposed change to the language consisting of the added support for primitive arbitrary precision numeric types `\bigint` and `\real` (Section 4)[1]. In this paper, we will call this new version of the language JMLa.
- An illustration of how JML currently fails to meet its design goals and objectives. This is achieved by presenting a selection of recently published JML specifications that are invalid or inconsistent under the current JML semantics (Section 5).
- An illustration of how JMLa better meets the JML design goals and objectives (Section 5). In particular, we show how JMLa: better matches user expectations, allows simpler specifications to be written and, gives ESC/Java the opportunity to detect more errors.

1.2 Significance of the research results

All of the cases of invalid or inconsistent specifications reported in Section 5 are from conference proceedings, books, or the main JML reference document. Hence, the authors and reviewers of these specifications certainly had confidence in their accuracy. We claim that these are not simply cases of erroneous specifications, but rather that they are an indication of the mismatch between the current JML semantics and the meaning specification readers and writers expect. We believe that JMLa better matches user expectations. To support our claim, we demonstrate that the invalidity and inconsistency of the cases disappear under JMLa semantics with either no syntactic changes or minor syntactic changes to the specifications. We also illustrate other benefits of JMLa—including safety, i.e. how it allows ESC/Java to detect more errors.

We believe it urgent to correct the JML language issues raised in this paper as the JML user base is increasing. Most importantly, it is being used in security-critical domains. For example, JML has been used for over two years as part of the VerifiCard project; this project's goal is to provide tools for the development of reliable Java Card based smart cards [BvdBJ02]. Of the cases discussed in Section 5, there are four published research results from the VerifiCard project—including one specification whose correctness is said to have been formally verified using the LOOP tool and PVS.

1.3 Overview

After a glance at JML's ancestry we take a closer look at its language design subgoals and the design decisions that ensued (Section 2). We then supplement these subgoals with a list of language design objectives inspired and partly adapted from those commonly used in the design of programming languages (Section 3). We introduce JMLa (Section 4) and the problematic JML specifications along with their resolution in JMLa (Sections 5). Finally, we discuss related work and conclusions (Sections 6 and 7).

2 JML language design goals and design choices

In this section we will be examining the language design subgoals of JML. To help us better understand the motivation behind the creation of these subgoals, we will consider the BISL languages that preceded JML, but before doing so we briefly review the nature of a BISL.

By definition, a BISL is tightly coupled to a particular programming language since its purpose is to allow developers to specify modules written in that programming language. A behavioral interface specification is a description of a module consisting of two main parts [Wing87]:

- an *interface*, that captures language specific elements that are exported by the module, such as field and method signatures;
- a *behavior*—as well as other properties and constraints—of the elements described in the interface.

[1] The need for slash characters in the names will also be explained in Section 4.

Prior to JML, the main BISLs were members of the Larch family of languages of which the two most notable members were Larch/C++ and LCL, the Larch/C interface specification language. A key characteristic of Larch is its two-tiered approach. The *shared* tier contains specifications written in the *Larch Shared Language* (LSL). These shared tier specifications, called traits, define multisorted first-order theories. The *interface* tier contains specifications written in a Larch interface language. Each interface language is specialized for use with a particular programming language, but all interface languages make use of LSL to express module behavior [GH93]. Unfortunately, the Larch languages were not widely adopted. One of the main reasons that has been cited is that the overhead of having to learn and use the Larch Shared Language is too large. Larch has nonetheless successfully given rise to *Splint*, an extended static checker for C currently in use by industry [EL02] and, a next generation of BISLs of which JML is a first instance.

JML inherits from Larch/C++ its general style of specification; the most important inherited language features are [LBR02]

- method specification through preconditions, postconditions and a frame axiom,
- model variables (fields) [Leino95].

As previously stated, JML's design has been guided by the overall goal that it be both practical and effective (**G0**). The JML Preliminary Design document further identifies these subgoals [LBR02]:

(**G1**) JML must be able to document the interfaces and behavior of existing Java software without change (regardless of the analysis and design methods used to create it).
(**G2**) JML should be readily understandable by Java programmers, including those with only standard mathematical training.
(**G3**) The language definition should be such that a formal semantics can be given and the language must be amenable to tool support.

While JML has directly inherited the most successful features of Larch, the following language design choice clearly demarcates it from its Larch predecessors [LBR02]:

(**D1**) Inspired by Eiffel [Meyer92], JML expressions are defined as an extension to side-effect free Java expressions[2]. Any expression that is valid in both languages is deemed to have the same meaning in both languages.

Hence, in JML the equivalent of the two Larch tiers have been merged. How then does one express in JML the mathematical theories that were codified in the Larch shared tier? By making use of the JML counterparts to Java interfaces and classes: *model* interfaces and *model* classes. These model interfaces and classes have essentially the same semantics as their Java counterparts but are used as an aide in writing specifications; i.e. they need not actually be implemented.

As a corollary to (**D1**), we note that

(**D2**) Support in JML for arbitrary precision integers is provided by means of the `JMLInfiniteInteger` model interface that documents methods somewhat like those of the standard Java `BigInteger` class.

We will illustrate use of `JMLInfiniteInteger` in Section 5.1. Note that currently there is no support for arbitrary precision floating-point numbers in JML.

3 Language design objectives

Designing a good computer language is a challenge. As a designer, one strives to achieve the right balance among often opposing design objectives. The challenge is particularly great for a BISL designer because a BISL is at the junction of two kinds of languages with significantly different purposes: the BISL, a *specification* language, and its underlying *programming* language. We follow Parnas and others in making the important distinction between behavioral software *descriptions* and their *specifications*. A behavioral specification is a statement of *requirements*, i.e. it is an abstraction that captures the *essence* of an entity's

[2] Side-effect free expressions are called "pure" expressions in JML.

behavior [Parnas01]. As BISL designers, we believe that it is particularly important to keep this distinction in mind.

We propose to evaluate JML and JMLa against the current JML design goals as well as the key language design objectives presented next. As an *overall objective* we believe that JML should be

- **Effective** for its intended purposes, i.e. it should:
  - Make it as easy as possible for developers to read and write *specifications* (in the sense of the term 'specification' previously given).
  - Be usable for run-time checking of assertions (including preconditions, postconditions and invariants).
  - Be usable for efficient extended static checking to detect as may errors as possible.
  - Be usable in the formal verification of Java modules.

(JML's run-time assertion checker compiler, ESC/Java and the LOOP tool currently support the last three bullets, respectively.) In support of **Effective**ness, we define the following objectives—the first four have been excerpted and adapted from a text on programming language design [Finkel96]:

- **Safe**. Semantic errors should be detectable, preferably using (extended) static checking.
- **Simple**. There should be as few basic concepts as possible. JML building upon Java, it should introduce as few new concepts as possible.
- **Clear**. Specification statements should be easy to read and understand.
- **Uniform**. Basic concepts should be applied consistently and universally.
- **Match User Expectations**, or to state this in another way, it should not "violate user expectations" [MC96].

The design decision to keep the semantics of JML and Java expressions the same, i.e. (**D1**), certainly maximizes **Simplicity**. It also seems like the most obvious way to make the notation readily understandable to Java programmers (**G2**), but, as we shall illustrate in Section 5, (**D1**) has had some unanticipated consequences.

4 JMLa

4.1 Definition

Syntactically, JMLa is identical to JML except for the introduction of two keywords `\bigint` and `\real`. Use of slash characters at the start of JML keywords is necessary for the support of language design goal (**G1**). The absence of slash characters would prevent us from writing specifications for existing Java code that made use of identifiers with the names `bigint` or `real`.

JMLa semantics differ from JML semantics not only because of the addition of the two new primitive types, but also in the meaning assigned to arithmetic expressions. In summary, JMLa semantics ensure that by *default*, numeric *operations* that can result in overflow are performed over arbitrary precision types. As is typical in Java, cast expressions can be used to override this default (and thus identify that the JML semantics apply). For example, unary plus has the same semantics in JMLa, JML and Java. Unary minus, on the other hand, will first promote an integer operand to `\bigint`, negate the operand value, and yield a result of type `\bigint`. Given that `i` is an `int`, then `-i` will be of type `\bigint` whereas `-(int)i` will denote an application of the JML (Java) unary minus and thus will be of type `int`. Similarly, `i+i` will be of type `\bigint`, and `i+(int)i` or, `(int)i+i` will be of type `int`. To preserve JML and Java semantics to the maximum extent possible, JMLa allows constant expressions to retain their JML semantics (regardless of the operators used) provided that at no point in the evaluation an overflow occurs. Therefore `-1+3` would be of type `int`, but `-2147483648` would be of type `\bigint`.

The semantics of JMLa and JML differ for at most the following operators: unary `-`, binary `+`, `-`, `*`, `/` and `%`. Since bit operators treat their operands as bit vectors, they are excluded from the list. We explain the semantics of JMLa relative to the ***differences*** between it and JML (and hence Java) by following the organization and section numbering of the second edition of the Java Language Specification [GJSB00].

4.2 Primitive Types and Values

The type `\bigint` is a primitive arbitrary precision integral type and `\real` is a primitive arbitrary precision floating-point type.

5.1.2 Widening Primitive Conversion

Widening primitive conversions are also supported from any integral type to `\bigint` and from any numeric type to `\real`. Widening primitive conversions preserve values exactly.

5.1.3 Narrowing Primitive Conversion

Narrowing primitive conversion shall also be supported from `\bigint` to any other integral type as well as from `\real` to any other numeric type. Like in Java, "a narrowing conversion may lose information about the overall magnitude of a numeric value and may also lose precision."

Conversion from `\bigint`. For an integer i, find a natural number m greater than 64 for which $|i| < 2^m$, and let j be the m-bit signed two's-complement representation of i. In this case, a narrowing conversion of i to an integral type T simply discards all but the n lowest order bits of j, where n is the number of bits used to represent type T.

Conversion from `\real` to `\bigint`. In a narrowing conversion of a real r, r is rounded to the nearest *integer* value (by rounding toward zero).

Conversion from `\real` to a (primitive numeric) type other than `\bigint`. In a narrowing conversion of a real r, the value r is first rounded to the nearest `double` value d, unless r is beyond the range of values of doubles in which case d will be `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` as appropriate. To complete the narrowing conversion of r one applies to d the Java rules for narrowing conversion of `double`s.

5.6 Numeric Promotions

5.6.1a Unary Numeric Promotion

When an operator applies unary numeric promotion to its operand (which must denote a value of a numeric type) the following rules apply, in order, using widening conversion (§5.1.2) to convert the operand as necessary:

- If the operand is not a cast expression and it is of a floating-point type, then it is converted to `\real`.
- Otherwise, if the operand is not a cast expression, then it is converted to `\bigint`.
- Otherwise, if the operand is of compile-time type `byte`, `short`, or `char`, it is converted to `int`.
- Otherwise, the operand remains as is and is not converted.

In any case, value set conversion (§5.1.8) is then applied.

5.6.2a Binary Numeric Promotion

When an operator applies binary numeric promotion to a pair of operands, each of which must denote a value of a numeric type, the following rules apply, in order, using widening conversion (§5.1.2) to convert operands as necessary:

- If neither operand is a cast expression and either is of a floating-point type, then both are converted to `\real`.
- Otherwise, if neither operand is a cast expression, then both are converted to `\bigint`.
- Otherwise, if either operand is of type `\bigint`, the other is converted to `\bigint`.
- Otherwise, if either operand is of type `\real`, the other is converted to `\real`.
- Otherwise, if either operand is of type `double`, the other is converted to `double`.
- Otherwise, if either operand is of type `float`, the other is converted to `float`.
- Otherwise, if either operand is of type `long`, the other is converted to `long`.

- Otherwise, both operands are converted to type `int`.

After the type conversion, if any, value set conversion (§5.1.8) is applied to each operand.

15 Expressions

15.15.4 Unary Minus Operator –

If the operand is not a constant expression, or it is a constant expression whose value cannot be represented by a `long`, then unary promotion is to be performed according to §5.6.1a. (Otherwise, promotion is carried out as described in §5.6.1).

15.17 Multiplicative Operators

Binary promotion for `*`, `/` and `%` is to be performed according to §5.6.2a unless the multiplicative expression is a constant expression whose value can be represented by a `long` (in which case promotion is performed according to §5.6.2).

15.18.2 Additive Operators (+ and -) for Numeric Types

Binary promotion is to be performed according to §5.6.2a unless the additive expression is a constant expression whose value can be represented by a `long` (in which case promotion is performed according to §5.6.2).

15.10 Array Creation Expressions

Dimension *expressions* can be of type `int`, `long` or `\bigint`. Hence, it remains a proof obligation to demonstrate that a dimension expression value is in the range `0` to `Integer.MAX_VALUE`. ***Arrays still contain at most*** `Integer.MAX_VALUE` elements, and the type of `length` is `int`.

15.13 Array Access Expressions

Index expression can be of type `int`, `long` or `\bigint`.

Thus, in effect, JMLa offers implicit promotion to `\bigint` and `\real` (under the circumstances described above). Although JMLa looses out on **Simplicity** when compared to JML (due to the introduction of new primitive types), it more than makes up for this loss by better meeting the other design objectives as we shall demonstrate in the Section 5.

4.2 Supporting class `JMLMath`

We also define a model class named `org.jmlspecs.lang.JMLMath` that, in particular, shall provide methods like those of `java.lang.Math` but that are defined over `\bigint`'s and `\real`'s.

5 Cases and consequences

In the subsections that follow we present seven cases of recently published JML specifications. All specifications, but one, are invalid or inconsistent under the current JML semantics. The one case with valid specifications (Section 5.5) is used to reinforce the idea that JMLa semantics are **Safer** and more **Uniform**.

5.1 Integer square root

The example in this section also serves as a basic introduction to method specifications in JML.

5.1.1 Case description (invalid and inconsistent specification)

The JML Preliminary Design document was the first, and it remains the principal document describing JML. First published in June of 1998, the document's opening example is a specification of an integer square root method like the one given in Figure 1 [LBR02]. The specification requires that a caller invoke

```
/*@ public normal_behavior
  @   requires y >= 0;
  @   ensures
  @          \result * \result <= y
  @       && y < (Math.abs(\result) + 1)
  @              * (Math.abs(\result) + 1);
  @*/
public static int isqrt(int y)
```

Figure 1. JML specification of `isqrt`

```
/*@ public normal_behavior
  @   requires y >= 0;
  @   ensures Math.abs(\result) <= y
  @       && \result * \result <= y
  @       && y < (Math.abs(\result) + 1)
  @              * (Math.abs(\result) + 1);
  @*/
public static int isqrt(int y)
```

Figure 2. "Corrected" JML specification of `isqrt`

the method with a nonnegative argument *y*, and in return, the method ensures that it will yield a resulting value, *r*, such that $r^2 \le y < (|r| + 1)^2$. The current definition of JML states that the expressions in the `requires` and `ensures` clauses are to be interpreted using the semantics of Java. At first sight, the specification may seem accurate but, under JML semantics, the specification allows `isqrt` to return (`Integer.MAX_VALUE` – 5) / 2, for example, when `y` is 1. This is due to Java's bounded integer arithmetic: e.g. the evaluation of `\result * \result` "overflows".

This unexpected situation is certainly not obvious on a first reading of the specification (violates **Clarity**[3]). Although rummaging through the Java language and API documentation *may* help clarify the issue, writing a small Java test program to print the value of the `ensures` clause expression is a quick and sure way to confirm it[4].The anomaly having been identified, the `ensures` clause of the specification was strengthened, in a subsequent edition of the JML Preliminary Design document, by adding the following conjunct: `Math.abs(\result) <= y`, see Figure 2. However, the "corrected" specification suffers from a similar anomaly in that it allows `Integer.MIN_VALUE` to be returned when `y` is 0. Surprisingly, these invalid specifications of `isqrt` remained uncontested for over four years even though they were published in the main JML reference document. After the author signaled this problem with the specification of Figure 2 [Chalin03], it was again published in a revised form, but problems remain. We note that all of the published versions of the JML `isqrt` specification are actually inconsistent. For example, when `y` is `Integer.MAX_VALUE`, the `ensures` clause is unsatisfiable (even though a valid answer of type `int` exists, namely 46340). It should not be this difficult to get such a simple specification right. From all of this specification churn, we deduce that JML experts and non-experts alike "read" into the specifications a meaning other than that provided by the current JML semantics (violates **User Expectations**). We believe that JML users generally think in terms of arbitrary precision arithmetic.

Since the problems in the specifications of `isqrt` were due to the use of bounded integer arithmetic, it would seem reasonable to attempt to rewrite the specification using JML's current language mechanism for arbitrary precision integers, i.e. the `JMLInfiniteInteger` model interface. Figure 3 is a version of the `isqrt` specification that makes use of `JMLInfiniteInteger`. The intent of the specification is obviously lost due to its verbosity, and it becomes clear why JML developers avoid using `JMLInfiniteInteger` (violates **Effectiveness**).

[3] Points in the text that give evidence of the violation of a language design objective will be indicated like this.
[4] Having to write *code* to confirm the *meaning* of such a simple *specification* seems counter intuitive.

```
/*@ public normal_behavior
  @  requires y >= 0;
  @  ensures
  @    (new JMLInfiniteInteger(\result)).abs().compareTo(
  @             new JMLInfiniteInteger(y)) <= 0
  @     && (new JMLInfiniteInteger(\result)).multiply(
  @             new JMLInfiniteInteger(\result)).compareTo(
  @                    new JMLInfiniteInteger(y)) <= 0
  @     && (new JMLInfiniteInteger(y).compareTo((new JMLInfiniteInteger(\result)).abs().
  @                  add(JMLInfiniteInteger.ONE).
  @                    multiply((new JMLInfiniteInteger(\result)).abs().
  @                              add(JMLInfiniteInteger.ONE))) < 0;
  @*/
public static int isqrt(int y)
```

Figure 3. Specification of `isqrt` using `JMLInfiniteInteger`

```
/*@ public normal_behavior
  @   requires y >= 0;
  @   ensures JMLMath.abs(\result) == JMLMath.floor(JMLMath.sqrt(y));
  @*/
public static int isqrt(int y)
```

Figure 4. JMLa specification of `isqrt` using `jmlspecs.lang.JMLMath`

5.1.2 JMLa: matching user expectations

Under the semantics of JMLa, the original specification of `isqrt` given in Figure 1 and its revised forms (in Figure 2 and [LBR02]) are valid and consistent (**Match**ing **User Expectations**).

5.1.3 JMLa: simpler, clearer, more effective

The integer square root method has been used as an opening example for presenting BISLs for over a decade. Invariably, the method has been described simply as: an integer approximation to the square root of y [GH93, LCPP, Leavens02a]. Being slightly more specific about the nature of the approximation, we can express our needed result with the mathematical formula: $\pm\lfloor\sqrt{y}\rfloor$. Using suitable methods from `jmlspecs.lang.JMLMath` we obtain the specification for `isqrt` given in Figure 4, which we believe to be **Simple**, **Clear** and **Effective**.

Can we achieve the same degree of clarity with JML? At best, we can write the following JML `ensures` clause expression by assuming the existence of a model class, `JMLInfiniteReal`, which axiomatizes real numbers:

```
ensures (new JMLInfiniteInteger(\result)).abs().compareTo(
                  JMLInfiniteReal.floor(JMLInfiniteReal.sqrt(y))) == 0;
```

This is less clear than the JMLa version, but more importantly, the axiomatization of reals numbers by means of `JMLInfiniteReal` would most likely require special provisions beyond the current JML semantics of model classes. (In JMLa, like in PVS, we can get around this difficulty by defining the semantics using a meta-language rather than in the language of JMLa.)

5.2 Decimal smart card class (two cases)

5.2.1 Description of cases (inconsistent specifications)

Smart card applets[5] have been identified as ideal candidates for the application of formal methods [KP03]. Two of the main reasons are that smart card applets are relatively small (therefore tractable) and, security concerns justify the extra effort required in applying rigorous or formal methods. In this section we comment on two recently published case studies in the specification and verification of part of a commercial smart card applet. The subject of the studies was `Decimal`, a small but key class of the Java Card Electronic Purse applet by Gemplus [Gemplus]. Although the authors of both studies began from the

[5] A smart card application is called an applet.

```
class Decimal {
  /*@ spec_public @*/ private short intPart, decPart;
  ...
  /*@ invariant PRECISION == 1000 &&
    @           -PRECISION < decPart && decPart < PRECISION;
    @
    @ normal_behavior
    @  requires   true;
    @  modifiable intPart, decPart;
    @  ensures    intPart == -\old(intPart) &&
    @             decPart == -\old(decPart) ...;
    @*/
  public Decimal oppose()
  ...
}
```

Figure 5. `Decimal` class specification excerpt

same `Decimal` class source, they proceeded independently to annotate it with JML specifications based on the available informal documentation. The studies differ in their approach to verification.

In one case, Cataño and Huisman, performed verification using ESC/Java [CH02]. The main benefit of ESC/Java is that verification is fully automated, but as can be expected, its verification is both unsound and incomplete. Like other similar tools (e.g. Splint [EL02]), it has nonetheless proven to be quite useful in detecting program errors. In the other case study, Breunesse *et. al.* made use of the LOOP tool and the PVS theorem prover to perform a complete and formal verification of the correctness of the code relative to its specification [BvdBJ02]. The LOOP tool was used to automatically translate Java code and JML specifications into the language of PVS. Correctness proofs were then performed within PVS using specially developed proof rules and tactics. In light of the complementary strengths of these tools, it has been suggested that they be used together by first applying ESC/Java to identify easily detectable (and often common) errors, and then subjecting the most critical aspects of the code to formal verification using the LOOP tool and PVS [BvdBJ02].

Consider the `Decimal` specification excerpt given in Figure 5. An instance of `Decimal` represents a fixed-point number with three digits of precision after the decimal point. Such a fixed-point number is implemented by two `short` fields: `intPart` for the integer part and `decPart` denoting the number of thousandths (e.g. 3 and 142, respectively, for the number 3.142). Note that the specification of `oppose` is inconsistent: i.e. there is a situation that satisfies its precondition (which is trivial since it is true) for which the postcondition is not satisfiable. This situation arises when `\old(intPart)`—the value of `intPart` in the pre-state, i.e. before `oppose` is called—is equal to `Short.MIN_VALUE`, in which case the first conjunct of the `ensures` clause would be evaluated as follows:

- `intPart == -\old(intPart)`
- `intPart == -(-32768`$_{\texttt{short}}$`)` substitution of the value of `\old(intPart)`, `Short.MIN_VALUE`.
- `intPart == -(-32768`$_{\texttt{int}}$`)` *numeric promotion* from `short` to `int` (due to unary minus semantics).
- `intPart == 32768`$_{\texttt{int}}$ application of unary minus (`int`).
- `(int)intPart == 32768`$_{\texttt{int}}$ *numeric promotion* of `intPart` from `short` to `int` (due to `==`).

There is no value that `intPart` can have that, after a widening primitive conversion to `int`, would make it equal to 32768 since `Short.MAX_VALUE` is 32767.

Interestingly, ESC/Java reports the possibility that the post condition of `oppose` may fail to be satisfied when `intPart` is `Short.MIN_VALUE` in the pre-state, but Breunesse *et. al.* make no mention of the problem being identified during the formal verification processing using LOOP and PVS. This may indicate that the LOOP semantic embedding of JML and/or Java into PVS does not accurately reflect the current JML or Java semantics, respectively.

5.2.2 JMLa: safer and more uniform semantics

Cataño and Huisman report that the use of ESC/Java allowed several errors to be detected. Of these errors, some[6] were detected only because `intPart` (and `decPart`) were declared to be of type `short`. As `int` is

[6] For example, the specification of `oppose` given in Figure 5 and that of `round` given in [CH02].

```
class Decimal {
 private short intPart, decPart;
 ...
 /*@ public model \real decimal;
   @ invariant decimal == round(decimal)
   @           decimal == intPart + decPart / (\real)PRECISION;
   @
   @  public normal_behavior
   @   ensures \result == JMLMath.nearestInteger(r * PRECISION) / (\real)PRECISION;
   @ public static model pure \real round(\real r);
   @
   @  public normal_behavior
   @   ensures \result == Short.MIN_VALUE - 0.999 <= r &&
   @                      r <= Short.MAX_VALUE + 0.999
   @ public static model pure \real inRange(\real r);
   @
   @*/
 ...
/*@ normal_behavior
  @  requires   d != null
  @             && inRange(round(decimal * d.decimal));
  @  modifiable decimal;
  @  ensures    decimal == round(\old(decimal) * d.decimal)
  @             && \result == this;
  @ also
  @ exceptional_behavior ...
  @*/
public Decimal mul(Decimal d) throws DecimalException
  ...
}
```

Figure 6. `Decimal` class specification using a `\real` model field

supported by the Java Card language [Sun02], it is conceivable that `intPart` could have been declared to be of type `int`. Consider, as we did previously, the evaluation in JML of the first conjunct of the `ensures` clause of `oppose` but assuming now that the type of `intPart` is `int`. In this situation:

- `intPart == -\old(intPart)`
- `intPart == -(-2147483648`$_{\texttt{int}}$`)` value of `\old(intPart)`, namely `Integer.MIN_VALUE`.
- `intPart == -2147483648`$_{\texttt{int}}$ application of unary minus[7].

Note that there are no widening primitive conversions. It is actually the presence of widening conversions that allowed ESC/Java to detect an anomaly when `intPart` was of type `short`.

Under JMLa semantics, even with `intPart` declared to be of type `int` we have:

- `intPart == -\old(intPart)`
- `intPart == -(-2147483648`$_{\texttt{int}}$`)` value of `\old(intPart)`.
- `intPart == -(-2147483648`$_{\texttt{\bigint}}$`)` *numeric promotion* from `int` to `\bigint`.
- `intPart == 2147483648`$_{\texttt{\bigint}}$ application of unary minus.
- `(\bigint)intPart == 2147483648`$_{\texttt{\bigint}}$ *numeric promotion*.

As before, there is no value that `intPart` can have that, after a widening primitive conversion to `\bigint`, would make it equal to `2147483648` since `Integer.MAX_VALUE` is `2147483647`. Hence, under JMLa semantics, ESC/Java would be as effective at detecting errors for expressions over `int`'s (or any integral numeric type) as it is for `short`'s or `byte`'s. This is not the case for JML: there is a non-uniform semantics for expressions of type `long` or `int` vs. `short` or `byte`.

5.2.3 JMLa: simpler, clearer, more effective

The availability of primitive arbitrary precision types allows us to write a specification for `Decimal` that is **Simpler** and **Clearer** than would otherwise be possible. In Figure 6, we illustrate a version of the `Decimal` specification in which we have introduced a `\real` model field named `decimal`. For each `Decimal` instance representing a fixed-point number *n*, `decimal` will be equal to *n*. Hence, `decimal` is always rounded to the appropriate number of digits of precision (as expressed in the invariant). The model class

[7] Note that `-Integer.MIN_VALUE` is equal to `Integer.MIN_VALUE`.

```
/*@ behavior
  @    requires d != null;
  @  modifiable intPart, decPart,
  @             DecimalException.instance,
  @             DecimalException.instance.ttype;
  @     ensures intPart >= 0 &&
  @             intPart * PRECISION + decPart ==
  @                 \old(intPart) * \old(d.intPart) * PRECISION
  @                 +
  @                 \old(intPart) * \old(d.decPart)
  @                 +
  @                 \old(decPart) * \old(d.intPart)
  @                 +
  @                 /// difficult rest-part, consisting of:
  @                 /// sign of product of decimal parts
  @                 (( (\old(d.decPart) >= 0 && \old(decPart) >= 0) ||
  @                    (\old(d.decPart) < 0 && \old(decPart) < 0) )
  @                       ? 1
  @                       : -1)
  @                 *
  @                 /// thousand-part of product of rounded decimal parts
  @                 (( (-100 <= \old(d.decPart) && \old(d.decPart) <= 100)
  @                         /// absolute value
  @                         ? ( (\old(d.decPart) >= 0)
  @                                 ? \old(d.decPart)
  @                                 : -\old(d.decPart) )
  @                         /// round last digit to 0 of absolute value
  @                         : (\old(d.decPart) >= 0)
  @                               ? 10 * (\old(d.decPart)/10)
  @                               : 10 * (-\old(d.decPart)/10) )
  @                    *
  @                    ((-100 <= \old(decPart) && \old(decPart) <= 100)
  @                         /// absolute value
  @                         ? ( (\old(decPart) >= 0)
  @                                 ? \old(decPart)
  @                                 : -\old(decPart) )
  @                         /// round last digit to 0 of absolute value
  @                         : (\old(decPart) >= 0)
  @                               ? 10 * (\old(decPart)/10)
  @                               : 10 * (-\old(decPart)/10) ) ) / 1000)
  @             &&
  @             DecimalException.instance
  @               == \old(DecimalException.instance) &&
  @             (DecimalException.instance != null ==>
  @                DecimalException.instance.ttype ==
  @                  \old(DecimalException.instance.ttype));
  @     signals (DecimalException e)
  @               intPart < 0 && ...
  @*/
public Decimal mul(Decimal d) throws DecimalException
```

Figure 7. Original specification of `mul(Decimal)`

method `round` makes use of the `jmlspecs.lang.JMLMath.nearestInteger` method[8]. Notice the conciseness and clarity of the specification of `mul` as compared to its original specification given in Figure 7.

Although expressions such as those of the `mul ensures` clause of Figure 7 may be useful as intermediate theorems to help checkers (like ESC/Java or the LOOP tool), they fail to server their purpose as *specifications* for human readers. This may indicate a need in JML for a language construct that would allow checker hints and/or theorems to be given. Such a construct should make it clear that these hints are not the specification.

5.3 Priority queue

5.3.1 Case description (inconsistent specification)

This section's example is taken from the main JML tutorial paper [LBR99b] cited on the JMLspecs.org documentation page—the sources for this example are also packaged with the JML distribution. Figure 8 is an excerpt from the specification of the `PriorityQueue` class. The method of interest to our presentation is `addEntry` that will add a new `QueueEntry` to a queue. The specification of the model class `QueueEntry` is given in Figure 9. JML model classes, like model fields and methods, need not be implemented; `QueueEntry` is used as an aid in specifying the behavior of methods like `addEntry` of the `PriorityQueue` class.

[8] Following the usual mathematical definition, `nearestInteger(r)` is easily specified as the integer nearest to its real argument `r` or, in the case that there are two such "nearest" integers, preference is given to the even one.

Unfortunately, under JML semantics, the specification of `addEntry` is inconsistent since it is possible for it to attempt to add a new `QueueEntry` with a negative time stamp. This would violate the `QueueEntry` constructor precondition: a negative argument could arise if `largestTimeStamp()` returned `Integer.MAX_VALUE` (since `Integer.MAX_VALUE + 1 == Integer.MIN_VALUE` which is negative).

```
public class PriorityQueue implements … {
…
 /*@   public normal_behavior
   @     requires   entries.isEmpty();
   @     assignable \nothing;
   @     ensures    \result == 0;
   @ also
   @   public normal_behavior
   @     requires   !(entries.isEmpty());
   @     assignable \nothing;
   @     ensures (\forall QueueEntry e;
   @                 entries.has(e);
   @                 \result >= e.timeStamp);
   @ public pure model int largestTimeStamp();
   @*/
 /*@  public normal_behavior
   @    requires argID != null
   @             && !contains(argID);
   @    assignable entries;
   @    ensures entries != null
   @     && entries.equals(
   @         \old(entries.insert(
   @            new QueueEntry(argID,
   @                argPriorityLevel,
   @                largestTimeStamp()+1))));
   @ also
   @   public exceptional_behavior …
   @*/
  public void addEntry(Object argID,
                       int argPriorityLevel)
```

Figure 8. `PriorityQueue.jml-refined` from org.jmlspecs.samples.jmlkluwer[9]

```
/*@ public pure model
  @ class QueueEntry implements JMLType {
  @
  @ public Object iD;
  @ public int priorityLevel;
  @ public int timeStamp;
  @
  @ public invariant iD != null
  @                  && timeStamp >= 0;
  @
  @  public normal_behavior
  @    requires … && argTimeStamp >= 0;
  @    assignable …, timeStamp;
  @    ensures … &&
  @            timeStamp == argTimeStamp;
  @
  @ public QueueEntry(Object argID,
  @                   int argLevel,
  @                   int argTimeStamp);
  @ ...
  @ }
  @*/
```

Figure 9. `QueueEntry` from org.jmlspecs.samples.jmlkluwer

5.3.2 JMLa: matching user expectation

There is no reason to place a bound on the values of time stamps as they are used solely in model classes. Therefore, changing the type of `timeStamp`, `argTimeStamp`, and `largestTimeStamp` from `int` to `\bigint` rids `PriorityQueue` of the inconsistency in the specification of `addEntry` without further modifications to the `PriorityQueue` and `QueueEntry` specifications.

5.4 MoneyOps interface

5.4.1 Case description (invalid specification)

This section highlights issues that arise in other specification taken from the JML Preliminary Design document [LBR02]. Figure 10 contains a very short excerpt from the `Money` interface specification; the only specification feature that is relevant to our discussion is the `long pennies` model field. `MoneyOps` (Figure 11) is a subinterface of `Money` that defines, among other methods, addition of `Money`'s by means of the `plus` method. The precondition of `plus` is defined in terms of the `can_add` model method, which in turn, makes use of `inRange`. Clearly `inRange` is intended to be true when its argument can be represented by a `long`. Unfortunately, `double`s have too few digits of precision to be able to represent all values of type `long`, thus, for example `inRange(Long.MAX_VALUE – 100)` is false when it should be true. Hence, under the current JML semantics, the specification of `plus` does not express the required behavior (failing to **Match User Expectations**).

[9] Note that the clause keywords `modifiable` and `assignable` are synonyms.

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

Figure 10. Money specification excerpt

```
public /*@ pure @*/ interface MoneyOps extends ...
{
  /*@  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
    @     requires m2 != null && can_add(m2);
    @     assignable \nothing;
    @     ensures \result != null
    @           && \result.pennies == this.pennies + m2.pennies;
    @*/
  public MoneyOps plus(Money m2);
  ...
}
```

Figure 11. `MoneyOps` specification excerpt

5.4.2 JMLa: matching user expectations

The specification of `plus` can be easily made to match user expectations by changing the all occurrences of `double`, in the specifications of `inRange`, `can_add` and `plus`, to either `\bigint` or `\real`.

5.5 Java Card API specifications

5.5.1 Case description ("fragile" validity and consistency)

In this section we illustrate how the non-uniform semantics of JML numeric expressions leads to specifications that have precarious validity or consistency. Poll, van den Berg and Jacobs have contributed to improving the documentation of Java Card API classes by specifying them with JML [PvdBJ00, PvdBJ01]. Consider the specification given in Figure 12 of the `arrayCopy` method from the `javacard.framework.Util` class [HP02]. It would seem quite reasonable for this method to be adapted[10] to support the copying of segments of arrays that are larger than 32K bytes—i.e. concretely, to have source and destination array offsets as well as `length` be of type `int` rather than `short`. Unfortunately, under JML semantics, making this likely type change renders the specification inconsistent: for example, sufficiently large values of `destOff` and `length` will satisfy the precondition but will result in the `ensures` clause subexpression `destOff+i` being negative in `dest[destOff+i]`. Of course, it might be possible to rewrite the `arrayCopy` specification to avoid this inconsistency, but no such rewriting is necessary under JMLa.

[10] Either within the same class, at some point in the future when Java Card hardware limitations have been relaxed, or when copied to another class to be reused outside the context of Java Card applets.

```
/*@ public behavior
   @    requires src != null &&  srcOff >= 0 &&  srcOff+length <= src.length
   @             && dest != null && destOff >= 0 && destOff+length <= dest.length
   @             && length >= 0;
   @  assignable dest[destOff..destOff+length-1], ...;
   @     ensures (\forall byte i; 0 <= i && i < length
   @                              ==> dest[destOff+i] == \old(src[srcOff+i]));
   @       ...
   @ also ...
   @*/
 public static final short
    arrayCopy(byte[] src, short  srcOff, byte[] dest, short  destOff, short  length)
 throws ...;
```

Figure 12. `javacard.framework.Util` class specification excerpt

5.5.2 JMLa: simpler, more uniform semantics

Under JMLa semantics, the `arrayCopy` specification is valid regardless of the particular integral types used in the declaration of `srcOff`, `destOff` and `length`. This would seem reasonable since the essence of the behavior of this method is to copy array segments, and this should be independent of the particular integral type assigned to the method arguments.

5.6 Other cases

There are other cases (e.g. from [RL00, PvdBJ01]) of invalid, inconsistent or "fragile" JML specifications with characteristics similar to those just presented. Likewise, JMLa semantics rids these other specifications of their invalidity or inconsistency with little or no syntactic changes to the specifications. We believe that the cases selected for detailed presentation in this paper provide a sufficient sampling.

6 Related work

6.1 Languages supporting primitive arbitrary precision numeric types

Several computer languages and tools provide basic language support for arbitrary precision integers including:

- **Specification languages**. Support for the integers is fundamental to most design specification languages, including:
  - Model based and algebraic languages such as B, OBJ, VDM, and Z [Bowen03].
  - **BISLs** such as Larch, via the Larch Shared Language, and Extended ML (EML), via its underlying programming language, ML.
- **Functional programming languages**: e.g. ML , Haskell , and various flavors of Lisp.
- **Tools**: e.g. proof tools such as PVS, HOL and Isabelle as well as numeric and symbolic mathematics systems such as Mathematica and Maple.

Extended ML (EML), is a BISL for Standard ML that adopts an approach similar to JML in that, as the name implies, EML is defined as an extension to ML and hence it subsumes its semantics—with the exception of imperative features [KST97]. EML does not suffer from the difficulties of JML described in this paper since ML integers are of arbitrary precision. It is certainly an interesting prospect to consider adding the equivalent of `\bigint` to Java instead of JML. Basic support for real numbers is most common in general design specification languages (such as those mentioned above) and less common in other languages. Symbolic mathematics packages often provide arbitrary precision rational numbers.

Certainly the fact that several other specification languages and tools offer basic support for arbitrary precision integers does not, in itself, justify their addition to JML. But, in this paper we have presented evidence that this is what JML users generally expect and hence, JML should support primitive arbitrary precision numeric types.

6.2 Assertion Languages

The problem highlighted in this paper can be generalized to other assertion-based languages such as Eiffel. In Eiffel, we speak of contracts rather than specifications. An Eiffel class contract consists of method contracts (given by means of requires and ensures clauses) as well as class invariants. Eiffel contracts are generally less semantically "rich" than JML specifications since all expressions in contracts are Eiffel expressions, and hence required to be executable. Consequently, it has been more difficult to find inconsistent or invalid Eiffel contracts, but they exist. As a simple example consider the contract for the `abs` functions from any one of the classes `INTEGER_8`, `INTEGER_16`, `INTEGER`, `INTEGER_64`, e.g.

```
abs: INTEGER_8 is
  ensure
    non_negative: Result >= 0
    same_absolute_value: (Result = item) or (Result = -item)
```

This contract is inconsistent since 'non_negative' cannot be satisfied when applied to `INTEGER_8.Max_value`. At this point, it is unclear to us how the solution presented here could be generalized to Eiffel.

6.3 Semantics

As previously mentioned, the LOOP tool translates JML specifications into the specification language of PVS by a "shallow" semantic embedding. Parts of the definition of this embedding have been published (e.g. [vdBJ01, JP01]) but they differ from the informal semantics of JML—as has been alluded to at the end of Section 5.2.1. It would seem that the semantics of JML expressions as treated in the LOOP tool more closely resembles the proposed semantics for JMLa. A more complete description of the semantic embedding is to be published soon as a Ph.D. thesis from the University of Nijmegen.

The work described in this paper has been inspired by our previous work on the semantics of LCL, the Larch/C interface specification language [CGR96]. In another report, we provide an exploration of language design alternatives for the JML support of arbitrary precision numeric types. In this same report, we provide a preliminary formal semantics of JMLa expressions as well as a comparison of the formal semantics of JML, JMLa, Larch/C++ and LCL [Chalin03].

7 Conclusions and future work

We believe that BISLs are one of the best ways of integrating formal methods into industrial practice [Chalin03]. Partial evidence of this is the increasing industrial use of Extended Static Checkers like Splint [EL02] and ESC/Java [Flanagan+02]. This paper focuses on the treatment of numeric types and the semantics of expressions over these types. This issue is fundamental to design specification languages since practically all specifications make use of numeric types. Although programming languages make the necessary compromise of providing support for bounded numeric types only, we have shown how a similar decision for a BISL like JML goes against user expectations. As a consequence we were able to illustrate several published JML specifications that were invalid or inconsistent. Seeking to better meet user expectations, we have defined a variant of JML called JMLa that has support for primitive arbitrary precision numeric types, and we have shown how

- JMLa semantics more closely match user expectations by demonstrating how invalid or inconsistent JML specifications recover their validity and consistency, when interpreted under JMLa with little or no changes to the specifications.
- JMLa can be used to write simpler, and clearer specifications.
- The meaning of JMLa specifications can be independent of the particular choice of numeric types of fields and variables (as it should be since, e.g. method specifications are meant to express *essential* method behavior which often is independent of field and variable types).
- ESC/Java will be able to detect more errors under JMLa semantics than it currently can for JML.

These points are particularly important as we witness the increased use of JML, particularly in security critical areas like smart cards. Of course these benefits come at the cost of a slightly more complex semantics and an increased departure from Java semantics. We believe though, that the benefits of JMLa outweigh its disadvantages.

In collaboration with other JML project partners we have begun transitioning ESC/Java and the JML tools to supporting JMLa. Preliminary results are encouraging since we have been able to use the tools to identify over a dozen other inconsistent method specifications in the JML model classes alone. The impact of supporting JMLa on the run-time assertion-checker compiler will be more significant. Although checking of `\bigint` expressions can be conveniently implemented using Java's `BigInteger` class we will still only be able to approximate `\real`'s using, say, `BigDecimal`.

We will also pursue our analysis of JML as other issues related to bounded vs. unbounded "data types" (such as arrays, sets and sequences) need to be explored further to ensure that there are no unsuspected consequences as there have been for numeric types. Our work on the language analysis and formalization of the semantics of JML will be pursued so as to progressively include more language elements.

8 Acknowledgments

We thank the anonymous referees, as well as Gary Leavens, Erik Poll and Peter Grogono for helpful comments on earlier revisions of this work.

9 References

[Bowen03] Jonathan Bowen, WWW Virtual Library: Formal Methods, http://www.afm.sbu.ac.uk. February 2003.

[BvdBJ02] C.-B. Breunesse, J. van den Berg, and B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, *AMAST'2002*, LNCS, pp. 304-318. Springer Verlag, 2002. Decimal class specification is available at www.cs.kun.nl/indexes/~ceesb/decimal/Decimal.java.

[CGR96] Patrice Chalin, Peter Grogono, and T. Radhakrishnan. "Identification of and solutions to shortcomings of LCL, a Larch/C interface specification language". In Marie-Claude Gaudel and James Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, LNCS 1051, pages 385–404. Formal Methods Europe, Springer, March 1996.

[CH02] N. Cataño and M. Huisman. Formal specification of Gemplus' electronic purse case study. Proceedings of Formal Methods Europe (FME 2002). LNCS 2391, pages 272-289. Springer, 2002.

[Chalin03] Patrice Chalin. *Back to Basics: Language Support and Semantics of Basic Infinite Integer Types in JML and Larch*. Technical Report 2002-003.3, Computer Science Department, Concordia University, April 2003. (Previous revisions: March 2003, October 2002.)

[EL02] David Evans and David Larochelle. Improving Security Using Extensible Lightweight Static Analysis. IEEE Software, Jan/Feb 2002.

[Finkel96] Raphael A. Finkel. *Advanced Programming Language Design*. Addison-Wesley, 1996.

[Flanagan+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In Cindy Norris and James B. Fenwick, editors, *Proceedings of Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of ACM SIGPLAN, pages 234–245, June 17–19 2002.

[Gemplus] Gemplus Purse applet. http://www.gemplus.com/smart/r_d/publications/case-study.

[GH93] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andr´es Modet, and Jeannette M. Wing.

[GJSB00] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java™ Language Specification. Second Edition, Addison-Wesley, 2000. Also java.sun.com/docs/books/jls/second_edition/html.

[HP02] Engelbert Hubbers and Erik Poll. jml.javacard.framework.Util.jml. University of Nijmegen, 2002. (www.cs.kun.nl/indexes/~erikpoll/publications/jc211_specs/jml/javacard/framework/Util.jml).

[JP01] Bart Jacobs and Erik Poll. A Logic for the Java Modeling Language JML. In: H. Hussmann (ed.), *Fundamental Approaches to Software Engineering* (*FASE*), LNCS 2029 pages284-299. Springer-Verlag 2001.

[KP03] Joseph Kiniry and Erik Poll. Opportunities and challenges for formal specification of Java programs. *Trusted Components Workshop*, Prato, Italy, January 2003.

[KST97] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML: A gentle introduction. *Theoretical Computer Science*, 173(2):445-484, 1997.

[LBR02] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University, TR #98-06t, December 2002.

[LBR99b] 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 of Businesses and Systems*, Chapter 12, pages 175-188. Kluwer, 1999.

[LCPP] Larch/C++ web page. www.cs.iastate.edu/~leavens/larchc++.html and www.cs.iastate.edu/~leavens/LarchC++.gif.

[Leavens+00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In OOPSLA '00 Companion, Minneapolis, Minnesota, pages 105-106.

[Leavens02a] Gary T. Leavens. *A Java Modeling Language*, slides from presentation given at Clemson University. May 31, 2002

[Leavens99] Gary T. Leavens. *Larch/C++ Reference Manual*, Iowa State University, Version 5.41, April 1999.

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

[MC96] L. McIver, and D. Conway. Seven Deadly Sins of Introductory Programming Language Design. *Proceedings, Software Engineering: Education & Practice 1996*, pages 309-316.

[Meyer92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[Owre96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger editors, Computer Aided Verification, LNCS 1102, pages 411-414. Springer, 1996.

[Parnas01] David Lorge Parnas. Description and Specification. In Daniel M. Hoffman and David M. Weiss editors. *Software Fundamentals: Collected Papers by David L. Parnas*, pages 1-6. Addison-Wesley, 2001.

[PvdBJ00] Erik Poll, Joachim van den Berg, Bart Jacobs. Specification of the JavaCard API in JML. *Fourth Smart Card Research and Advanced Application IFIP Conference (CARDIS'2000)*, 2000.

[PvdBJ01] Erik Poll, Joachim van den Berg, Bart Jacobs. Formal Specification of the JavaCard API in JML: the APDU class. *Computer Networks*, Volume 36, Issue 4, pp. 407-421, Elsevier Science, 2001.

[RL00] Clyde Ruby and Gary T. Leavens. Safely Creating Correct Subclasses without Seeing Superclass Code. In *OOPSLA 2000 Conference Proceedings*, pages 208-228. Volume 35, number 10 of ACM SIGPLAN Notices, Oct. 2000.

[Sun02] Java Card 2.2 Virtual Machine Specification. Sun Microsystems. May 13, 2002.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In: T. Margaria and W. Yi editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS),* LNCS 2031, pages 299-312. Springer, 2001.

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