JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 6
Special issue: ECOOP 2003 workshop on FTfJP

JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics

Patrice Chalin, Concordia University, Canada

Abstract

The Java Modeling Language (JML) is a notation for specifying and describing the detailed design and implementation of Java modules. An important language design goal of JML has been to preserve the semantics of Java to the extent possible. Thus, in particular, Java numeric expressions have the same meaning in JML. We illustrate how such a semantics fails to match the expectations of specification authors and readers who generally think in terms of arbitrary precision arithmetic (rather than the fixed precision provided by Java). As a result, an unusually high number of published JML specifications are invalid or inconsistent, including cases from the security critical area of smart card applications. We briefly examine JML's ancestry and language design principles; this helps to explain the origin of the semantic gap between user expectations and the current meaning given to JML numeric expressions. With the objective of better matching user expectations we introduce JMLb, a variant of JML supporting primitive arbitrary precision numeric types as well as "math modes" to control the semantics of arithmetic expressions. This is done in a manner that is consistent with JML's language design goals. A semantics of JMLb expressions is given by means of an embedding into PVS. The problem presented here will arise in the design of most interface specification languages that must deal with, e.g., mathematical integers in specifications and their fix precision approximations in code. We examine how the problem may manifest itself in other languages (such as Eiffel, Spark and the UML/OCL-Java notation of the KeY project) and comment on the applicability of our solution.

1 INTRODUCTION

The Java Modeling Language (JML) is a notation for specifying and describing the detailed design and implementation of Java modules [LBR99]. 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. An important language design goal of JML has been to preserve the semantics of Java to the extent

Cite this article as follows: Patrice Chalin: "JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics", in *Journal of Object Technology*, vol. 3, no. 6, Special issue: ECOOP 2003 workshop on FTfJP, June 2004, pp. 57-79.
http://www.jot.fm/issues/issue_2004_06/article3

possible. Thus, in particular, Java numeric expressions have the same meaning in when they occur in JML specifications. We illustrate how such a semantics fails to match the expectations of specification authors and readers who generally think in terms of arbitrary precision arithmetic (rather than the fixed precision provided by Java). As a result, an unusually high number of published JML specifications are invalid or inconsistent, including cases from the security critical area of smart card applications [Chalin03].

In this article we briefly describe JML's ancestry and language design principles (Section 2). This will help to explain the origin of the semantic gap between user expectations and the current meaning given to JML numeric expressions. With the objective of better matching user expectations, we introduce JMLb (and its predecessor JMLa), as variants of JML supporting *primitive* arbitrary precision numeric types as well as "math modes" to control the semantics of arithmetic expressions (Sections 3 and 4). This is done in a manner that is consistent with JML's language design goals and objectives [Chalin03]. A formal semantics of JMLb expressions is given (Section 5) as well as an example of its application. We note that the problem presented here will arise in the design of most interface specification languages which must deal with, e.g., mathematical integers in specifications and their fix precision approximations in code. We examine how the problem may manifest itself in other languages (such as Eiffel, Spark and the UML/OCL of KeY) and comment on the applicability of our solution. Other related and future work are also discussed (Sections 6 and 7).

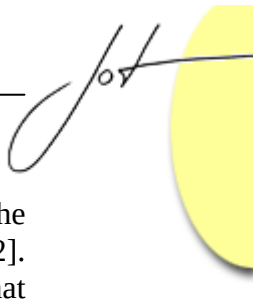
2 JML

Ancestry and language design principles

JML is a Behavioral Interface Specification Language (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 *behavioral* description (including other properties and constraints) of the interface elements.

Prior to JML, the main BISLs were members of the Larch family of languages of which two notable members are Larch/C++ [Leavens99] and LCL, the Larch/C interface specification language [GH93]. 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]. In a departure from the Larch tradition, Leavens *et. al.* have defined JML as a single-tier BISL [LBR02]. Experience with Larch/C++ lead to the opinion that having to learn two—somewhat disparate—languages (C++ and LSL) in order to be able to read and write specifications, was too big a hurdle to overcome for most developers. The design intent has been to make JML a superset of (sequential) Java. A key language design principle of JML has been to preserve the semantics of Java to the extent possible: that is, if a phrase is valid in Java and JML, then it should have the same meaning in both languages. Adherence to this principle should greatly reduce the burden required to learn, understand and use JML.

As is often the case for language design principles, their benefits come at a cost. Java was designed as a programming language, not a specification language. Although JML builds upon Java by adding language constructs for the purpose of expressing specifications, it remains that core Java phrase sets, like expressions, are (for the most part) shared by both languages. This renders expression semantics more complex than, for example, in Larch. Furthermore, as we shall see in the following section, developers are in a different mindset when reading or writing specifications, particularly when it comes to reasoning about integer arithmetic.

```
/*@ 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 1. JML specification of `isqrt(int)`

A semantic gap, motivating examples

Consider the specification in Figure 1 of an integer square root method, `isqrt`; it was excerpted from the June 2002 edition of the main JML reference document [LBR02]. The specification requires that a caller invoke the method with a nonnegative argument *y*, and in return, the method ensures that it will yield a value, *r*, such that: $|r| \leq y \ \wedge \ r^2 \leq y < (|r| + 1)^2$. The current definition of JML states that the expressions in the `requires` and `ensures` clauses of Figure 1 are to be interpreted using the semantics of Java. As a consequence (and a simple Java prototype will justify this claim), a valid implementation of `isqrt` would be permitted to return `Integer.MIN_VALUE` when *y* is 0. This unexpected situation arises because Java integral types have a fixed precision and because operators over these types obey rules of modular arithmetic—thus, for example

```
Integer.MIN_VALUE == Integer.MAX_VALUE + 1
Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE
Integer.MIN_VALUE * Integer.MIN_VALUE == 0
(Integer.MIN_VALUE + 1) * (Integer.MIN_VALUE + 1) == 1
```

```
/*@ spec_public */
private short intPart, decPart;

/*@ normal_behavior
  @  modifiable intPart, decPart;
  @  ensures    intPart == -\old(intPart) &&
  @             decPart == -\old(decPart) ...;
  @*/
public Decimal oppose()
```

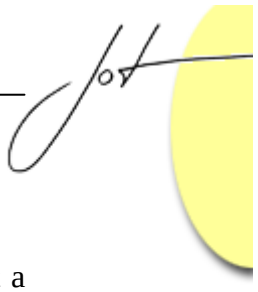
Figure 2. `Decimal` class specification excerpt

As another example, consider the specification given in Figure 2 which was excerpted from a paper on the formal verification of an electronic purse applet [BvdBJ02]. We show part of the `Decimal` class specification: 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)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.

What has gone wrong? These JML specifications (and others) demonstrate that specifiers most often ignore the finiteness of numeric types. (Incidentally, this is also true of Java programmers [BS04].) Stated positively, specifiers generally *think* in terms of arbitrary precision arithmetic when they read and write specifications. A survey, including the two cases just described, is given in [Chalin03] of invalid and inconsistent JML specifications caused by this problem. Hence, there is a semantic gap between user expectations and the current language design and semantics of JML numeric types. As a way of leading up to our proposed means of bridging the semantic gap, we explore next how the specification of `isqrt` might be "fixed" within the current semantics of JML.



Attempting to mend the gap

The `isqrt` specification can easily be corrected so that `Integer.MIN_VALUE` is not a valid result by ensuring that arithmetic overflow does not occur while interpreting the `ensures` clause expression. A strengthened specification is given in Figure 3—differences relative to the previous specification are underlined. (It should be noted that one of our goals here is to preserve the overall *form* of the ensures clause predicate while exploring means of adapting or adorning the predicate so that its meaning matches our expectations.) Explicit type widening ensures that all operators will be applied to `long` arguments.

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

Figure 3. JML specification of `isqrt(int)` with cast to `long`

Although explicit type casting solves the problem in this particular case, it would be ineffective if the argument and return types of `isqrt` were changed from `int` to `long`. The specification of this new `isqrt(long)` method can none-the-less be corrected by making use of the only available mechanism in JML to express arbitrary precision arithmetic, namely, the `JMLInfiniteInteger` model class. The resulting specification of `isqrt(long)` is given Figure 4. Notice how we might have gained accuracy, but we loose significantly in clarity. The intent of the specification is obviously lost due to its verbosity, and it becomes clear why JML developers might avoid using `JMLInfiniteInteger` for something as common as expressions involving arithmetic.

```
/*@ 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 4. Specification of `isqrt(long)` using `JMLInfiniteInteger`

We come to the conclusion that there is no general and practical language mechanism in JML that would allow us to mend the semantic gap. Hence, in the next two sections we explore JML language variants named JMLa and JMLb. They represent our approaches to closing the semantic gap while, at the same time, balancing JML's language design goals. By presenting an intermediate step (JMLa) towards our final solution (JMLb) we can better convey the motivation behind our choice of language features.

3 JMLA: PRIMITIVE ARBITRARY PRECISION NUMERIC TYPES

Shortening the semantic gap

The overly verbose specification of `isqrt(long)` defined using the `JMLInfiniteInteger` reference type makes it obvious that, just as Java has primitive fixed precision numeric *value* types, JML should have primitive arbitrary precision numeric value types. To this end we introduce in JMLa the primitive numeric types `\bigint` and `\real` representing arbitrary precision integers and floating point numbers, respectively. Like other JML keywords that can occur in expressions, these start with a slash character so as to prevent name clashes in specifications for existing Java code that made use of identifiers with the names `bigint` or `real`.

External to the language we also define a model class named `org.jmlspecs.lang.JMLMath` that, in particular, provides methods like those of `java.lang.Math` but that are defined over `\bigint`'s and `\real`'s. Like in Java, all specifications implicitly import `org.jmlspecs.lang.*`.

A JMLa specification for `isqrt(long)` is given in Figure 5. Note how it preserves the clarity and the form of the original specification while achieving the required degree of accuracy.

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

Figure 5. JML specification of `isqrt(long)` with casts to `\bigint`

Closing the semantic gap

Most specifiers think in terms of arbitrary precision arithmetic, yet the semantics of expressions in Java and JML are such that fixed precision arithmetic is the *default* interpretation. Introducing new primitive arbitrary precision types to the JML language is

one step towards narrowing this gap, but it does not close it. Alternatives for closing the gap include:

- Adapting the Java language to support a primitive arbitrary precision integral type, as is done in languages like ML and Haskell. Such a new language feature would naturally become a part of JML. Of course, offering programmatic support for reals is not possible.
- Adding distinct operator names (e.g. \oplus or ⊕) to distinguish between operations over fixed vs. arbitrary precision numeric types. In specifications, Java operators would refer to operations over `\bigint` and `\real` whereas the special operators would refer to Java's fixed precision arithmetic. Although this solution is common in Larch, it would go against JML's main language design goals.
- Adding implicit promotion to `\bigint` for integral expressions. (Of the examples that we have surveyed, we have yet to justify the need for implicit promotion to `\real`.)

Among these alternatives the last would appear to be the most feasible and the one that clashes the *least* with the language design goals of JML and hence it is chosen for JMLa.

Informal semantics

JMLa introduces the primitive types `\bigint` and `\real`, and appropriately places them as new "top" elements of the Java numeric type hierarchy as illustrated in Figure 6. Type numeric widening and narrowing are defined as a natural extension of the rules of Java.

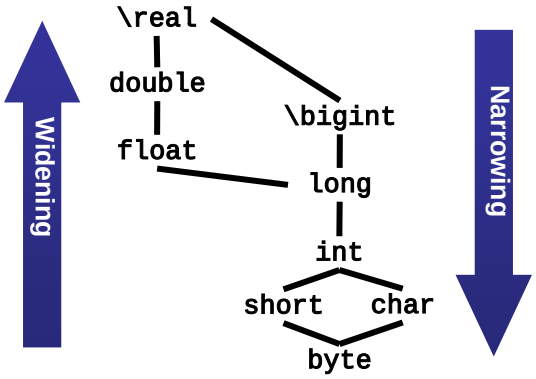


Figure 6. JMLa primitive numeric type hierarchy

With respect to the semantics of integral arithmetic expressions, in JMLa we ensure that numeric operations that can cause overflow are performed over `\bigint` by default. We call the operators that can result in overflow *unsafe* operators; they are: unary `-`, binary `+`, `-`, * and `/`. Early in our design of JMLa, expression semantics followed a simple rule: all unsafe operators unconditionally promoted their integral operands to `\bigint` before performing the operation. This turned out to be impractical since Java programs contain many instances of constant expressions involving unsafe operators, the most common of which is `-1`. By the simple semantic rule, `-1` would be implicitly promoted to `\bigint` and if this expression was on the right-hand side of an assignment or the argument to a

method call, then it would most likely have to be *explicitly* cast back a fixed precision type. Such explicit casts are unnecessary because it is easy to determine statically whether the evaluation of a constant expression will result in overflow or not. Thus we amended the rule to preserve Java semantics if: the operands are constant expressions and operator evaluation does not result in overflow.

JMLa Expression	Equivalent JMLa Expression (all conversions made explicit)	Result Type	Same semantics as in Java?
`+i`	`+i`	`int`	Yes, since unary + is safe.
`-i`	`-(\bigint)i`	`\bigint`	No, unary – is unsafe.
`-5`	`-5`	`int`	Yes, since –5 is an `int` value.
`-Integer.MIN_VALUE`	`-(\bigint)Integer.MIN_VALUE`	`\bigint`	No, since the constant expression value is not in the range of `int`.
`i + j`	`(\bigint)i + (\bigint)j`	`\bigint`	No, since binary + is unsafe.
`Integer.MIN_VALUE - 1`	`(\bigint)Integer.MIN_VALUE - (\bigint)1`	`\bigint`	No, since the expression value is not in the range of `int`.
`2 + bi`	`(\bigint)2 + bi`	`\bigint`	Almost: Java-like type promotion
`2 + (int)bi`	`(int)2 + (int)bi`	`int`	Yes.
`i * f`	`(float)i * f`	`float`	Yes, no implicit promotion to `\real`.
`3 * 5 – Short.MAX_VALUE`	`3 * 5 – (int)Short.MAX_VALUE`	`int`	Yes (const. expr. value is an `int`).
`d / r`	`(\real)d / r`	`\real`	Almost: Java-like type promotion
`(\bigint)d / 2`	`(\bigint)d / (\bigint)2`	`\bigint`	Almost: Java-like type promotion

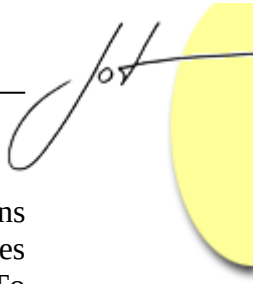
Figure 7. Sample JMLa expressions
(assume `int i,j; \bigint bi; float f; double d; \real r`)

Examples of JMLa expressions are given in Figure 7. Notice how `-5` has type `int` whereas `-Integer.MIN_VALUE` has type `\bigint` because evaluation of the latter constant expression in Java would result in an overflow. Other JMLa languages features were also defined [Chalin03] but since they are not a part of JMLb they are not discussed here.

4 JMLB: MATH MODES

Better balance of language design goals

In all cases that we have encountered of invalid or inconsistent JML specifications (due to fixed precision arithmetic), the cases recover their validity and consistency under JMLa with either no syntactic changes or minor syntactic changes to the specifications [Chalin03]. Thus, the semantic gap has been closed, but this is achieved at the cost of contravening one of the basic design goals of JML, namely, that expressions that are valid in Java and JML should have the same meaning. At a minimum, there should be a



visual cue for specification readers to indicate that the semantics of numeric expressions are not like in Java. It is with this idea that we introduced the notion of arithmetic modes in JMLb. The default mode in JMLb coincides with Java semantics (like in JML). To indicate that JMLa semantics are in effect one must explicitly provide a modifier either to a class or a method declaration. We believe that this allows JMLb to achieve a better balance of the JML language design goals with a small extra cost (as compared to JMLa).

Math modes

In JMLb, there are three integral arithmetic modes, or *math modes* for short. The semantics of expressions for each mode is as follows:

- *Java math* corresponds to Java semantics.
- *Bigint math* corresponds to the JMLa implicit promotion semantics.
- *Safe math* is like Java math except that arithmetic overflows are signaled by means of exceptions (like C# checked mode).

To set the math mode for all specification expressions in a class one annotates the class definition with one of the following modifiers: `spec_java_math`, `spec_bigint_math` and `spec_safe_math`. These modifiers can also be applied to individual method definitions (examples will follow shortly). The scope of a modifier is the entire declaration to which it is applied. For finer grained control JMLb has operators that limit the scope of a mode to a given expression, *E*: e.g. `\java_math(E)`, `\bigint_math(E)`, and `\safe_math(E)`.

A sample JMLb specification is given in Figure 8. In this specification, the class modifier `spec_bigint_math` informs us that all specification expressions in the class are to be interpreted in bigint math mode by default. The first method specification is a slightly modified version of the specification of `isqrt` given in Figure 1 in which we have replaced the occurrences of `Math` by `JMLMath`. Notice that under JMLb, the specification of `isqrt` is valid since the expressions are interpreted over `\bigint` rather than `int`. The second specification is of an increment method that demonstrates the use of a math mode modifier, as applied to a method declaration, as well as a math mode operator. In this case, we make use of `\java_math` to specify that `i+1` should be interpreted as in Java so that, e.g., `i+1` will be equal to `Integer.MIN_VALUE` when `i` is equal to `Integer.MAX_VALUE` (as indicated in the given specification example). The final method specification illustrates the use of `\bigint` in a model method, thus defining `inc` as equivalent to the successor function over the infinite set of mathematical integers.

When unspecified, the default mode is Java math, but to ensure that JML users are aware of the possible consequences of this default, JML tools will issue a warning if the math mode is not explicitly stated. In most cases, JML authors will want to choose `spec_bigint_math`.

JMLb also provides the modifiers `code_java_math`, `code_bigint_math` and `code_safe_math` that allow the semantics of arithmetic expressions to be changed in Java code. Of course, for this to be effective one must use special compilers such as the

MultiJava or the JML Run-time Assertion Checker (RAC) compilers [Burdy+03]. We believed that this can be an convenient means of providing arbitrary precision arithmetic (for bigint math) or run-time overflow detection (for safe math). The latter feature is built in to the C# language and is called checked mode. Like in JMLb, the default mode in C# is unchecked. These code math modes are not discussed any further in this article.

```
public /*@spec_bigint_math@*/ class JMLbSample
{
  /*@ public normal_behavior
    @   requires y >= 0;
    @   assignable \nothing;
    @   ensures JMLMath.abs(\result) <= y  &&
    @           \result * \result <= y  &&
    @           y < (JMLMath.abs(\result) + 1)
    @                * (JMLMath.abs(\result) + 1);
    @*/
  public static int isqrt(int y) {
    return (int) Math.sqrt(y);
  }

  /*@ public normal_behavior
    @   assignable \nothing;
    @   ensures \result == \java_math(i + 1);
    @ for_example
    @   public normal_example
    @     requires i == Integer.MAX_VALUE;
    @     assignable \nothing;
    @     ensures \result == Integer.MIN_VALUE;
    @*/
  /*@spec_safe_math@*/
  public static int increment_and_wrap(int i) {
      return i+1;
  }

  /*@ public normal_behavior
    @   assignable \nothing;
    @   ensures \result == i + 1;
    @*//*@
  @ public pure model static \bigint inc(\bigint i) {
  @    return i+1;
  @ }
  @*/
}
```

Figure 8. Sample JMLb specification

Advantages over JML

Some of the key advantages of JMLb over JML include:

- JMLb semantics more closely match user expectations. We demonstrate in [Chalin03] how all of the invalid or inconsistent JML specifications given in that article recover their validity and consistency when interpreted under JMLa (i.e. JMLb with the `spec_bigint_math` modifier applied to each class) with little or no changes to the specifications.
- JMLb can be used to write simpler, and clearer specifications (as compared, e.g., to use of `JMLInfiniteInteger)`.
- The meaning of JMLb specifications in bigint math mode can be independent of the particular choice of numeric type 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. This is not the case in Java and JML; e.g. "`i ==` *E*" may be unsatisfiable if the identifier `i` is declared to be of type `short`.
- ESC/Java [Flanagan+02] will be able to detect more errors under JMLb semantics than it currently can for JML (in particular due to the previous point).
- Verification proofs will be greatly simplified as we recover the familiar laws of arithmetic when operating over `\bigint` and `\real` (e.g. associativity, commutativity and closure of operators).

These points are particularly important as we witness the increased use of JML, especially 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 JMLb outweigh its disadvantages.

5 JMLB SEMANTICS

The LOOP tool, developed at the University of Nijmegen, provides a semantics of Java and JML by means of a shallow embedding into PVS [vdBJ01, JP03]. PVS, short for Prototype Verification System, is the name given to a powerful theorem prover and to the specification language that it supports [PVS]. Following the LOOP approach, this section also presents a formalization of JMLb semantics by means of an embedding into PVS. We focus only on those aspects of JMLb that differ from JML—namely the semantics of arithmetic expressions under the various math modes—while ignoring important issues, such as abnormal termination in expressions, which are already effectively handled in LOOP. The semantics given here can be regarded as complementary to the LOOP semantics, eventually to be integrated with it (more will be said of this in the conclusion).

Abstract syntax and semantic objects

The semantics of JMLb expressions is defined by means of an "inference system" in a style referred to as natural semantics [Winskel93]. The inference rules allow us to establish the validity of *elaboration predicates* of the form

$\rho \text{ H } a \xrightarrow{A} x$

where A is generally the name of an abstract syntax phrase class. Such a predicate asserts that the syntactic object a corresponds to the semantic object x under the context ρ; we will also say "a elaborates to x under ρ." For the cases covered here, the context will be an environment containing the declarations under which elaboration is to be performed, a will be a JMLb expression and x a PVS expression qualified with its type.

$e \in \text{EXPR} ::= c_\tau \mid \iota \mid op(e_1, \ldots, e_k) \mid (\tau)\, e \mid \texttt{\old}(e) \mid e.\iota \mid e.\iota\,(e_1, \ldots, e_k) \mid (q\ \tau\ \iota\ ;\ e)$
$\quad \mid \texttt{\java\_math}(e) \mid \texttt{\bigint\_math}(e) \mid \texttt{\safe\_math}(e) \mid \ldots$
$\tau \in \text{TYPENM} ::= \iota \mid \ldots$
$op \in \text{OPNM}$
$q \in \text{QUANTNM} ::= \texttt{\forall} \mid \texttt{\exists}$

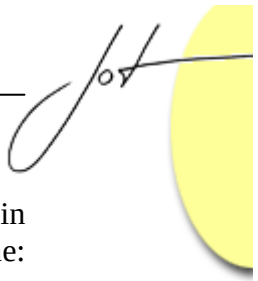
Figure 9. Abstract syntax of JMLa expressions

The abstract syntax for expressions relevant to our presentation is given in Figure 9. The defined cases are:

- An integral literal constant of type $\tau \in \{\texttt{int}, \texttt{long}\}$.
- An identifier representing a logical variable (including `\result`, method parameters and quantifier variables). Since our focus in this article is on the particularities of JMLb semantics of expressions over numeric types, we shall make the simplifying assumption that all class and instance members are expressed in the form $e.\iota$ so as to be distinguishable from the occurrence of a logical variable.
- An operator applied to one or more arguments. Operators include those of Java (e.g. +, `-`, *) and JML (e.g. `==>`, `<==>`).
- A type cast expression.
- A pre-state expression.
- A field access expression.
- A method invocation expression. (Recall that methods in JML expressions must be "pure" [LBR02].)
- A quantified expression.
- Math mode expressions.

JMLb expressions are translated into the "semantic objects" of PVS expressions, whose *annotated* abstract syntax is

$\varepsilon \in \text{PVSEXPR} ::= c : \tau \mid op(\varepsilon_1, \ldots, \varepsilon_k) : \tau \mid q\,(\iota : \tau) : \varepsilon$



Each PVS expression is annotated with its type. This allows us to ensure that, in particular, overloaded operators can be disambiguated. The cases of PVSEXPR define: constants, operators (including logical connectives and equality), quantified formulas (where q is either FORALL or EXISTS). Elaboration of expressions is done in the context of an environment, $\rho \in$ ENV that can be thought of as a mapping from identifiers into their attributes. The following identifiers have a special meaning:

- `\result` is the JML logical variable that can be used in ensures clauses to denote the value returned by a method;
- `\mathMode` denotes the "currently active" math mode;
- `\state` denotes the evaluation state context and is either bound to `pre` or `post`. By default, requires clause expressions are evaluated in `pre` and ensures clause expressions in `post`.

The updated environment denoted by $\rho \oplus \{ \iota \Box \alpha \}$ is the same as ρ except that it maps ι to α. Note that in the initial JMLb environment ρ_0, `\mathMode` is set to `java`.

Primitive numeric types in PVS

Before presenting the elaboration rules, we will explain how JMLb primitive numeric types are modeled in PVS. The JMLb arbitrary precision types `\bigint` and `\real` are modeled by the standard PVS types `integer` and `real`. For convenience, we have also defined a synonym for `integer` named `bigint`. We have created simple theories, all of the same form, for each of the bounded precision integral types. As an example, an excerpt of the theory for `int` is given in Figure 10. Notice how the `int` type is simply defined as the subtype of `integer` that contains values in the range `min` to `max` inclusive. A key function in this theory is `narrow` which effectively defines narrowing primitive conversion to `int`. All arithmetic operators are defined using their `integer` counterparts followed by an application of `narrow`. Thus, addition of `int`'s is defined as the addition of their values interpreted as `integer`'s followed by a narrowing of the result to `int`: i.e. `add(i,j) = narrow((i:int + j:int):integer):int`. Bitwise operators are tricky to handle in PVS hence, for the most part, we use the bit vector library that is part of the standard distribution of PVS.

```
int: THEORY
 BEGIN
  IMPORTING number_theory@mod_nt, div@div,
            div@div_alt, div@rem
  ...
  twoPn: posint = 4294967296
  max  : nat    = 2147483647
  min  : negint = -2147483648

  int: TYPE+ = {i: integer | min <= i AND i <= max}
                 CONTAINING 0;

  % Narrowing primitive conversion to int
  narrow(i: integer): int =
      LET b:nat = mod(i, twoPn) IN
        IF b <= max THEN b ELSE b - twoPn ENDIF

  neg(i:int)  : int = narrow(-i)
  add(i,j:int): int = narrow(i + j)
  sub(i,j:int): int = narrow(i - j)
  mul(i,j:int): int = narrow(i * j)

  % Division rounds towards zero (JLS2.0, Section 15.17.2):
  %
  div(i:int, j:{j:int|j /= 0}): int = narrow(div.div(i,j))

  % Remainder satisfies (i/j)*j + (i%j) == i for all
  % values except j = 0, including i = max and j = -1
  % (JLS2.0, Section 15.17.3).
  %
  rem(i:int, j:{j:int|j/=0}): int = narrow(rem.rem(i,j))

  div_rem: LEMMA FORALL (i,j:int):
      j /= 0 => div(i,j)*j + rem(i,j) = i
  ...

  bit_neg(i:int): int = -i - 1
  ...
 END int
```

Figure 10. PVS theory for `int`

Elaboration rules

Elaboration rules for JMLb expressions are given in Figure 11. For each syntactic case of EXPR we have combined the type and expression elaborations into a single rule. Aside from the backslash prefixing the JMLb type names `\bigint` and `\real`, JMLb and PVS type names coincide, hence we will make no distinction between them, using the same

type name τ in both the abstract syntax and PVS expressions. The given semantics do not cover rules for constant expressions, and (as was mentioned earlier) it ignores issues of abnormal termination.

Literals and Logical Variables

$$\frac{}{\rho \vdash c_\tau \longrightarrow c : \tau}\ \text{Literal} \qquad \frac{\rho(\text{LVAR}\ \iota) = \tau}{\rho \vdash \iota \longrightarrow \iota : \tau}\ \text{Logical Var}$$

Operators

$$\frac{\begin{array}{c} op \in \mathit{unsafeOps} \wedge \rho(\texttt{\textbackslash mathMode}) = \texttt{bigint} \\ \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\tau_1, ..., \tau_k) \in \mathit{ArgType}_{op} \qquad \mathit{JML.conv}_{op}(\tau_1, ..., \tau_k) = (\tau'_1, ..., \tau'_k) \\ \mathit{resultType}_{op} = \tau \qquad (\tau'_1, ..., \tau'_k \to \tau) \in \rho(\text{OP}\ op).\mathit{types} \end{array}}{\rho \vdash op(e_1, ..., e_k) \longrightarrow \phi_{op}(\epsilon_1, ..., \epsilon_k) : \tau}\ \text{Op}$$

$$\frac{\begin{array}{c} op \notin \mathit{unsafeOps} \vee \rho(\texttt{\textbackslash mathMode}) \neq \texttt{bigint} \\ \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\tau_1, ..., \tau_k) \in \mathit{ArgType}_{op} \qquad \mathit{Java.conv}_{op}(\tau_1, ..., \tau_k) = (\tau'_1, ..., \tau'_k) \\ \mathit{resultType}_{op} = \tau \qquad (\tau'_1, ..., \tau'_k \to \tau) \in \rho(\text{OP}\ op).\mathit{types} \end{array}}{\rho \vdash op(e_1, ..., e_k) \longrightarrow \phi_{op}(\epsilon_1, ..., \epsilon_k) : \tau}\ \text{OpJava}$$

Math Mode Operators

$$\frac{\rho \oplus \{\ \texttt{\textbackslash mathMode} \mapsto m\ \} \vdash e \longrightarrow \epsilon : \tau}{\rho \vdash \texttt{\textbackslash}m\texttt{\_math}(e) \longrightarrow \epsilon : \tau}\ m\text{\_math}$$

Casts

$$\frac{\rho \vdash e \longrightarrow \epsilon : \tau' \qquad \tau' \leq \tau}{\rho \vdash (\tau)e \longrightarrow \epsilon : \tau}\ \text{Widen} \qquad \frac{\rho \vdash e \longrightarrow \epsilon : \tau' \qquad \tau' > \tau}{\rho \vdash (\tau)e \longrightarrow \texttt{narrow}(\epsilon) : \tau}\ \text{Narrow}$$

Old and Quantifier Expressions

$$\frac{\rho \oplus \{\ \texttt{\textbackslash state} \mapsto \texttt{pre}\ \} \vdash e \longrightarrow \epsilon : \tau}{\rho \vdash \texttt{\textbackslash old}(e) \longrightarrow \epsilon : \tau}\ \text{old} \qquad \frac{\begin{array}{c} \rho \oplus \{\ \text{LVAR}\ \iota \mapsto \tau\ \} \vdash e \longrightarrow \epsilon : \texttt{boolean} \\ \tau \in \mathit{PrimitiveNumeric} \end{array}}{\rho \vdash (q\ \tau\ \iota\ e) \longrightarrow (q\ \ (\iota : \tau) : \epsilon) : \texttt{boolean}}\ \text{Quant}$$

Static Field/Method Access

$$\frac{\begin{array}{c} (\text{CLASS}\ \iota') \in \operatorname{dom} \rho \\ \mathit{lookup}_{tp}(\rho, \iota', \iota) = \tau \\ \sigma = \rho(\texttt{\textbackslash state}) \end{array}}{\rho \vdash \iota'.\iota \longrightarrow \texttt{val}(\sigma,\ \iota'.\iota) : \tau}\ \text{Static Field} \qquad \frac{\begin{array}{c} \rho \vdash e_i \longrightarrow \epsilon_i : \tau_i \quad \text{for } i = 1..k \\ (\text{CLASS}\ \iota') \in \rho \\ \mathit{lookup}_{tp}(\rho, \iota', \iota) = [\tau_1, ..., \tau_k \to \tau'] \end{array}}{\rho \vdash \iota'.\iota(e_1, ..., e_k) \longrightarrow \iota'.\iota(\epsilon_1, ..., \epsilon_k) : \tau'}\ \text{Static Method}$$

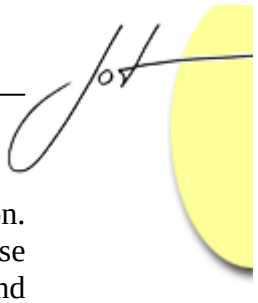
Figure 11. JMLb expression semantics, selected rules

- The rules for literals [Literal] and logical variables [Logical Var] illustrate that they are translated almost literally. Thus, `3` and `5L` elaborate to `3:int` and `5:long` respectively. Similarly, provided that a logical variable has been declared in scope, it elaborates to the same name qualified with its declared type (of course `\result` would have to be mapped to a special PVS name). Method parameters, quantifier variables and `\result` are modeled as logical variables.

Operator(s) *op*	**Argument Type(s)** $ArgType_{op}$	**Argument Conversion** $conv_{op}$	**Result Type** $resultType_{op}$	**PVS Function(s)** ϕ_{op}
`+` (unary)	Numeric	*unp* to τ	τ	`id`: τ → τ
`-` (unary)	Numeric	*unp* to τ	τ	`τ.neg`
`~`	Integral	*unp* to τ	τ	`τ.bit_neg`
`!`	`boolean`	none	`boolean`	`NOT`
`*, /, %`	Numeric, Numeric	*bnp* to (τ,τ)	τ	`τ.mul, τ.div, τ.rem`
`+, -`	Numeric, Numeric	*bnp* to (τ,τ)	τ	`τ.add, τ.sub`
`<<, >>, >>>`	Integral, Integral	arg1:*unp* to τ arg2: *unp*	τ	`τ.lshift, τ.rshift, τ.rshift_u`
`<, <=, >, >=`	Numeric, Numeric	*bnp*	`boolean`	`<, <=, >, >=`
`==, !=`	Numeric, Numeric	*bnp*	`boolean`	`==, /=`
	`boolean, boolean`	none	`boolean`	`==, /=`
`&, ^, |`	`boolean, boolean`	none	`boolean`	`AND, XOR, OR`
	Integral, Integral	*bnp* to (τ,τ)	τ	`τ.bit_and, τ.bit_xor, τ.bit_or`
`&&, ||`	`boolean, boolean`	none	`boolean`	`AND, OR`
`==>`	`boolean, boolean`	none	`boolean`	`==>`
`<==>, <=!=>`	`boolean, boolean`	none	`boolean`	`==, /=`

Figure 12. Semantics of selected JMLb operators

- There are two elaboration rules for expressions involving operators. [Op] applies to all unsafe operators (these are listed in Section 3) while in bigint math mode. The [OpJava] rule applies to operators that are not unsafe, or to any operator while in Java or safe math mode. To apply either of these rules one must make use of the information provided in Figure 12. Let *op* be an operator that appears in column 1 of the table, then the remaining columns define: the required argument type(s) ($ArgType_{op}$), the kind of argument conversion to be applied ($conv_{op}$), the resulting type of the expression ($resultType_{op}$) and finally, the PVS function corresponding to the operator *op* (ϕ_{op}). In the argument type column: "numeric" includes Java numeric types as well as `\bigint` and `\real`; "integral" includes Java integral types as well as `\bigint`. The argument conversions that can be



applied correspond to unary numeric promotion or binary numeric promotion. These promotion functions come in two variants (see Figure 13): those corresponding to implicit promotion to arbitrary precision types (*JML*.*) and those that imitate the standard Java promotion rules (*Java*.*).

- The math mode operators merely set the active mode in the environment under which their arguments are interpreted.
- Type casts are processed in two cases depending on the nature of the cast: i.e. whether it corresponds to a widening [Widen] or narrowing primitive conversion [Narrow]. Type widening requires no special operator in PVS. On the other hand, narrowing to type τ requires the application of the `narrow` function defined in the theory of τ.
- Interpretation of a pre-state expression is simply achieved by changing the elaboration context with respect to the current state variable (`\state`). For the given rules, this currently has an impact only on field member access.
- Quantified expressions are translated almost directly into PVS. The JML quantifiers `\forall` and `\exists` are named `FORALL` and `EXISTS` in PVS.
- Field access makes use of the function `val` which is defined over the state held in ρ. Given a reference *r* of type `Ref[τ]`, and a state *s*, then `val`(*s*,*r*) will yield the value of type τ contained in *r*.
- Member access is restricted to pure methods—i.e. in particular, methods that are guaranteed not to have side-effects.

JML.unp(τ) =
  if τ ∈ { `float`, `double`, `\real` }
  then `\real`
  else `\bigint` ***end***

JML.bnp(τ1, τ2) = ***if*** τ1 or τ2 is one of
  { `float`, `double`, `\real` }
  then (`\real`, `\real`)
  else (`\bigint`, `\bigint`) ***end***

Java.unp(τ) = ***if*** τ ∈ { `byte`, `short`, `char` }
  then `int` ***else*** τ ***end***

Java.bnp(τ1, τ2) = ***if*** `\real` ∈ {τ1, τ2} ***then*** (`\real`, `\real`)
  else-if `\bigint` ∈ {τ1, τ2} ***then*** (`\bigint`, `\bigint`)
  else-if `double` ∈ {τ1, τ2} ***then*** (`double`, `double`)
  else-if `float` ∈ {τ1, τ2} ***then*** (`float`, `float`)
  else-if `long` ∈ {τ1, τ2} ***then*** (`long`, `long`)
  else (`int` , `int`) ***end***

Figure 13. JMLb type conversion functions

Example

As an example of the application of the elaboration rules we will use the JMLb `isqrt` specification given in Figure 8. The resulting PVS translation is shown in Figure 14. Notice how the PVS expressions closely resemble their JMLb counterparts. As a partial indication of the suitability of the semantic translation, we have been successful in proving the consistency of the `isqrt` specification (making use of PVS `abs` for `JMLMath.abs`).

```
IMPORTING int, ...

isqrt_requires(y: int): bool =  y >= 0;
isqrt_ensures(y, result: int): bool =
  JMLMath.abs(result) <= y AND result * result <= y AND
      y < (JMLMath.abs(result)+1) * (JMLMath.abs(result)+1);

isqrt_consistent: LEMMA FORALL (y:int):
  EXISTS (result:int):
    isqrt_requires(y) => isqrt_ensures(y,result)
```

Figure 14. PVS definition of `isqrt`

6 RELATED WORK

Primitive arbitrary precision numeric types

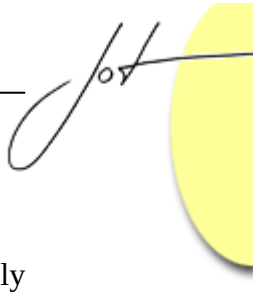
Several computer languages and tools provide basic language support for arbitrary precision integers including: specification languages, such as B, OBJ, VDM, and Z [Bowen03]; BISLs such as Larch, and Extended ML (EML); Functional languages like ML, Haskell, and Lisp; proof tools like PVS and symbolic mathematics systems such as Mathematica and Maple. Basic support for real numbers is most common in general design specification languages and proof tools and less common in other languages. Symbolic mathematics packages often provide arbitrary precision rational numbers.

Eiffel

Eiffel, well know for its promotion of design by contract, also makes use of pre- and post-conditions in "method" specifications [Meyer92]. Like JML, Eiffel does not have support for arbitrary precision integers (as part of its kernel), but it is subject to similar problems due to the use of fixed precision arithmetic types in specifications. As an example, consider the following specification for the `abs` function as taken from any one of the kernel classes `INTEGER_8`, `INTEGER_16`, `INTEGER`, or `INTEGER_64`:

```
abs: INTEGER_16 is
  ensure
    non_negative: Result >= 0
    same_absolute_value: (Result = item) or (Result = -item)
```

The `non_negative` clause cannot be satisfied when applied to `INTEGER_16.Min_value`. Due to Eiffel's type system and language semantics with respect to numeric conversions and method/operator resolution, it is unclear how the solutions presented here could be generalized to Eiffel.



Spark

Spark is a carefully chosen subset of Ada suitable for use in the development of highly reliable software [Barnes03]. Tool support includes the Spark Examiner which can perform extended static checking of Spark code annotated with assertions (e.g. subprogram pre- and post-conditions, loop invariants).

Although integral types are of fixed precision in Spark (and Ada), Spark does not suffer from the same problems as JML because integral arithmetic in specification expressions is interpreted over the arbitrary precision integers. Thus, the following Spark function specification

```
function Negate(X: Integer)
--# returns -X;
```

would result in the generation of the verification condition `Integer'First <= -X <= Integer'Last` which is unprovable when `X` is `Integer'First`. The same result would be obtained in `spec_bigint_math` mode in JMLb—in fact, Spark specification expression semantics corresponds to JMLb `spec_bigint_math` mode. The main difference is that there is no type corresponding to the mathematical integers in Spark. Arithmetic in Ada programs is checked: i.e. overflows are reported by means of exceptions. Hence, Ada code is interpreted in the equivalent of JMLb's code safe math mode. Spark does not currently support specification and reasoning about exceptions.

UML/OCL and Java in KeY

The KeY project (http://www.key-project.org) offers tool support for the specification and verification of Java Card programs. Specifications are expressed in UML/OCL and verification is carried out in Dynamic Logic [Ahrendt+04]. For the purpose of program verification in KeY, Java is extended with four primitive arbitrary precision types (called *arithmetical* types): `arithByte`, `arithShort`, `arithInt` and `arithLong`. Arithmetic operators are defined (for each arithmetical type) with a semantics identical to their Java counterparts except in those situations where overflow would occur; in these cases, the semantics of the operators over the arithmetical types is left unspecified. It should be noted that this Java language extension is only used during the verification process (and hence, need not be supported by, say, a specially adapted Java compiler).

The KeY approach to specification and verification (with respect to integral arithmetic) is the following [BS04, Section 3.5]:

1. Specifications are written in UML/OCL using the OCL type `Integer` which corresponds to the mathematical integers.
2. Implementations of operations (methods) specified using the `Integer` type must be declared with one of the KeY arithmetical types.
3. Verification is then performed and if successful, then the semantics of KeY guarantee that the implementation will preserve its validity if all arithmetical types are replaced by their corresponding Java types.

The KeY approach is similar to Larch in that it appears to have two tiers (see Section 2). The shared or mathematical tier is provided by UML/OCL. As a consequence, the semantic gap that was present in JML is absent from KeY because specification statements are expressed in UML/OCL using the mathematical integers. On the other hand, there is no clearly identified interface specification language in KeY. This can give rise to problems: for example, consider the following UML/OCL method specification:

```
context C::negate(x: Integer) : Integer
post result = -x
```

From this specification we can deduce that acceptable implementations will consist of a class named `C` exporting a method named `negate`, but the signature of `negate` is not fixed; any of the following (among others) might be acceptable: `byte negate(byte)` or `int negate(int)` or even `int negate(byte)`. No such ambiguity is possible in JML since the method signature is fixed in the "interface" part of the specification. We believe that this might explain, in part, why the KeY solution (e.g. adding for new primitive types to Java) is more complex than the approach adopted in JMLb (in which we added only one new integral type).

7 CONCLUSIONS AND FUTURE WORK

We have illustrated a semantic gap between user expectations of the meaning of expressions over numeric types and the current JML language definition. Due to this gap, several published JML specifications are invalid or inconsistent—we have presented two such problematic specifications. To better meet user expectations, we have defined a variant of JML called JMLb that has support for primitive arbitrary precision numeric types `\bigint` and `\real`. JMLb also introduces arithmetic modes and allows specifiers to select the mode that is most appropriate for the specification at hand, generally though, it will be `spec_bigint_math` mode. A semantics of JMLb expressions is given and its application is illustrated by means of a simple example.

Members of Concordia's Dependable Software Research Group (DSRG) have completed the implementation of JMLb in the JML checker and this has allowed us to detect or confirm over two dozen inconsistent or erroneous JML specifications. Work is also progressing towards inclusion of JMLb support (both spec and code math modes) in `jmlc`, the JML runtime assertion checker compiler (RACC) as well as support for the code math modes in the MultiJava compiler.

Cees-Bart Breunesse and Joe Kiniry from the University of Nijmegen have nearly completed the integration of JMLb in the LOOP tool and ESC/Java2, respectively. Hence, it is now possible to use the LOOP tool to perform verifications of JML annotated Java modules under JMLb semantics. The main task that remains to be done for ESC/Java2 is finding a suitable replacement for the Simplify theorem prover which inadequately handles arbitrary precision integral arithmetic.

8 ACKNOWLEDGMENTS

We thank the anonymous referees whose detailed comments that have contributed to improving this article. We also thank members of the JML community for discussions on JMLb, particularly Erik Poll and Joe Kiniry. Thanks to Frederic Rioux for his contribution to the implementation of JMLb support in the JML checker.

9 REFERENCES

[Ahrendt+04] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. "The KeY Tool". In *Software and Systems Modeling*, 2004, to appear.

[Barnes03] John Barnes. *High Integrity Software: The Spark Approach to Safety and Security*. Addison-Wesley 2003.

[BS04] Bernhard Beckert, Steffen Schlager. "Software Verification with Integrated Data Type Refinement for Integer Arithmetic". In *Fourth International Conference on Integrated Formal Methods, Proceedings. LNCS 2083*. Springer-Verlag, 2004.

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

[Burdy+03] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. "An overview of JML tools and applications". In *Eighth International Workshop on Formal Methods for Industrial Critical Systems (FMICS '03)*, pp. 73-89. Volume 80 of Electronic Notes in Theoretical Computer Science, (http://www.elsevier.nl/locate/entcs) Elsevier, June, 2003.

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

[Chalin02] Patrice Chalin. "Back to Basics: Language Support and Semantics of Basic Infinite Integer Types in JML and Larch". (http://www.cs.concordia.ca/~faculty/chalin/papers/TR-2002-003.pdf) *Technical Report* 2002-003.3, Computer Science Department, Concordia University, October 2002, revised March, May 2003.

[Chalin03] Patrice Chalin. "Improving JML: For a Safer and More Effective Language". In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli (Eds.), *FME 2003 (*http://matrix.iei.pi.cnr.it/FMT/FME*), International Symposium of Formal*

Methods Europe, Pisa, Italy, Sept. 8-14, 2003, Proceedings. LNCS 2805 Springer-Verlag 2003. http://www.fmeurope.org/

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

[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, Andres Modet, and Jeannette M. Wing.

[JP03] Bart Jacobs and Erik Poll. Java Program Verification at Nijmegen: Developments and Perspective. In Proceedings of the International Symposium on Software Security (ISSS) 2003, to appear. Also, University of Nijmegen Technical Report NIII-R0316. http://www.cs.kun.nl/~erikpoll/publications/loop_history.html

[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. http://www.cs.iastate.edu/~leavens/JML/prelimdesign/prelimdesign_toc.html

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. "JML: A notation for detailed design". In *Behavioral Specifications of Business and Systems*, pages 175-188. Kluwer, 1999.

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

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

[PVS] The PVS Specification and Verification System. http://pvs.csl.sri.com.

[vdBJ01] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, LNCS 2031, pages 299-312. Springer, 2001. http://www.cs.kun.nl/~bart/PAPERS/tacas01.ps.Z.

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

[Winskel93] Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction*. Foundations of Computing Series. MIT Press, 1993.

About the author

Patrice Chalin is an Assistant Professor in the Department of Computer Science and Software Engineering of Concordia University. He is head of the Dependable Software Research Group (DSRG), conducting research on the language design, semantics and tool support for specification and programming languages. He can be reached at chalin@cs.concordia.ca. See also http://www.cs.concordia.ca/~chalin.