Java Program Verification Challenges

B.P.F. Jacobs, J.R. Kiniry, M.E. Warnier

Nijmegen Institute for Computing and Information Sciences/

NIII-R0310 March 2003

Nijmegen Institute for Computing and Information Sciences
Faculty of Science
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Java Program Verification Challenges *

Bart Jacobs, Joseph Kiniry, Martijn Warnier

Dep. Comp. Sci., Univ. Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.
`{bart,kiniry,warnier}@cs.kun.nl`
`http://www.cs.kun.nl/~{bart,kiniry,warnier}`

Abstract. This paper aims to raise the level of verification challenges by presenting a collection of sequential Java programs with correctness annotations formulated in JML. The emphasis lies more on the underlying semantical issues than on verification.
Keywords: Java, program specification, verification
Classification: 68Q55, 68Q60 (AMS'00); D.1.5, D.2.4, F.3.1 (CR'98).

1 Introduction

In the study of (sequential) program verification one usually encounters the same examples over and over again (e.g., stacks, lists, alternating bit protocol, sorting functions, etc.), often going back to classic texts like [11, 14, 15]. These examples typically use an abstract programming language with only a few constructs, and the logic for expressing the program properties (or specifications) is some variation on first order logic. While these abstract formalisms were useful at the time for explaining the main ideas in this field, they are not very helpful today when it comes to actual program verification for modern programming and specification languages. The aim of this paper is to give an updated collection of program verification examples. They are formulated in Java [13], and use the specification language JML [19, 21].

The examples presented below are based on our experience with the LOOP tool [3] over the past five years. Although the examples have actually been verified, the particular verification technology based on the LOOP tool does not play an important rôle: we shall focus on the underlying semantical issues. The examples may in principle also be verified via another approach, like those of Jack [6], Jive [23], Krakatoa [10], and KeY [2].

We shall indicate when static checking with the ESC/Java [12] tool brings up interesting semantical issues in our examples[1]. When ESC/Java checks an annotated program, it returns one of three results. The result "passed" indicates that ESC/Java believes the implementation of a method fulfills its specification; in that case we'll say ESC/Java accepts the input. A result of "warning" indicates that an error potentially exists in the program but ESC/Java was

* To appear in the LNCS proceedings of FMCO 2002.
[1] We use the final binary release (v. 1.2.4, 27 September 2001) of ESC/Java.

unable to prove its existence definitively (e.g., a counter-example or instruction trace). Finally, the message "error" indicates a specific bug in the program, typically a potential run-time violation like a `NullPointerException` or an `ArrayIndexOutOfBoundsException`.

Of course the examples below do not cover all possible topics. For instance, floating point computations and multi-threading[2] are not covered. In general, our work has focused on Java for smart cards and several examples stem from that area. Other examples are taken from work of colleagues [4], from earlier publications, or from test sets that we have been using internally. They cover a reasonable part of the semantics of Java and JML. Most of the examples can be translated easily in other programming languages. The examples do not involve semantical controversies, as discussed for instance in [5].

These examples are not meant to be canonical or proscriptive; they simply give a flavor of the level of completeness (and thus, complexity) necessary to cover modern programming language semantics.

The verification examples are organised in the following categories.

- Control flow and side-effects
- Overflow and bitwise operations
- Static initialization
- Inheritance
- Non-termination
- Specification issues

The organization of the paper roughly follows this categorization. Each example is accompanied by a short explanation of why the example is interesting and the challenges involved. But first we briefly describe the specification language JML.

2 JML

The Java Modeling Language, JML [19], is a behavioral interface specification language designed to specify Java modules. It can be used for classes as a whole, via class invariants and constraints, and for the individual methods of a class, via method specifications consisting of pre-, post- and frame-conditions (assignable clauses). In particular, it is also possible within a method specification to describe if a particular exception may occur and which post-condition results in that case.

JML annotations are to be understood as predicates, which should hold for the associated Java code. These annotations are included in the Java source files as special comments indicated by `//@`, or enclosed between `/*@` and `*/`. They are recognised by special tools like the JML run-time checker [9], the LOOP compiler, and the Krakatoa verification condition generator.

Class invariants and constraints are described as follows.

```
/*@                              /*@
  @ invariant <predicate>          @ constraint <relation>
  @*/                              @*/
```

[2] For a *meta*-theory on multithreaded Java programs see [1] in this volume.

An invariant is thus a predicate on the underlying state space. It must hold after termination of constructors, and also after termination (both normal and exceptional) of methods, provided it holds before. Thus, invariants are implicitly added to postconditions of methods and constructors, and to preconditions of normal (non-constructor) methods. A constraint is a relation between two states, expressing what should hold about the pre-state and post-state of all methods. In this paper we use no constraints, and only one invariant (in Subsection 3.6). However, in practice they contain important information.

Next we give an example JML method specification of some method `m()`.

```
/*@ behavior
  @      requires <precondition>;
  @  assignable <items that can be modified>;
  @       ensures <normal postcondition>;
  @       signals (E) <exceptional postcondition>;
  @*/
public void m()
```

Such method specifications may be understood as an extension of correctness triples $\{P\}m\{Q\}$ used in Hoare logic, because they allow both normal and exceptional termination. Moreover, the postconditions in JML are relations, because the pre-state, indicated by `\old(--)`, may occur. We shall see many method specifications below.

JML is intended to be usable by Java programmers. Its syntax is therefore very much like Java. However, it has a few additional keywords, such as `==>` (for implication), `\old` (for evaluation in the pre-state), `\result` (for the return value of a method, if any), and `\forall` and `\exists` (for quantification).

This paper will not pay much attention to the semantics of JML (interested readers should see [21] for such). Hopefully, most of the JML assertions are self-explanatory. However, there are three points that we would like to mention.

- In principle, expressions within assertions (such as an array access `a[i]`) may throw exceptions. The JML approach, see [21], is to turn such exceptions into arbitrary values. Of course, one should try to avoid such exceptions by including appropriate requirements. For instance, for the expression `a[i]` one should add `a != null && i >= 0 && i < a.length`, in case this is not already clear from the context. This is what we shall always do.
- JML uses the subtype semantics for inheritance, see [22]. This means that overriding methods in subclasses should still satisfy the specifications of the overridden ancestors in superclasses. This is a non-trivial restriction, but one which is essential in reasoning about methods in an object-oriented setting. However, it does not hold for all our examples (see for instance Subsection 3.8). In that case we simply write no specification at all for the relevant methods.
- JML method specifications form proof obligations. But also, once proved, they can be used in correctness proofs of other methods. In that case one first has to establish the precondition and invariant of the method that is called, and subsequently one can use the postcondition in the remainder of

the verification (which will rely heavily on the called method's assignable clause).

An alternative approach is to reason not with the specification, but with the implementation of the method that is called[3]. Basically, this means that the body of the called method gets substituted at the appropriate place. However, this may lead to duplication of verification work, and makes proofs more vulnerable to implementation changes. But if no specification is available (see the previous point), one may be forced to reason with the implementation.

In the examples below we shall see illustrations of method calls which are used both by specification and by implementation.

For readers unfamiliar with JML, this paper may hopefully serve as an introduction via examples. More advanced use of JML in specifying API-components may be found for instance in [24][4].

3 Verification challenges

This section describes our Java+JML examples in several subsections. Our explanations focus on the (semantic) issues involved, and not so much on the actual code snippets. They should be relatively self-explanatory.

3.1 Aliasing and field access

Our first example, seen in Figure 1, might seem trivial to some readers. The return expression of the method `Alias.m()` references the value of the field `i` of the object `c` value via an aliased reference to itself in the field `a`. We present this example because it represents (in our view) the bare minimum necessary to model a language like Java. ESC/Java has no problem verifying this program. Either the implementation or specification of the constructor of class `C` can be used to verify method `Alias.m()`.

3.2 Side-effects in expressions

One of the most common abstractions in program verification is to omit side-effects from expressions in the programming language. This is a serious restriction. Figure 2 contains a nice and simple example from [4] where such side-effects play a crucial rôle, in combination with the logical operators. Recall that in Java there are two disjunctions (`|` and `||`) and two conjunctions (`&` and `&&`). The double versions (`||` and `&&`) are the so-called conditional operators: their second argument is only evaluated if the first one is false (for `||`) or true (for `&&`).

[3] This only works if one actually knows the run-time type of the object on which the method is called.

[4] See also on the web at `www.cs.kun.nl/~erikpoll/publications/jc211_specs.html` for a specification of the Java API for smart cards.

```
class C {
    C a;
    int i;

    /*@ normal_behavior
      @     requires true;
      @   assignable a, i;
      @      ensures a == null && i == 1;
      @*/
    C() { a = null; i = 1; }
}

class Alias {
    /*@ normal_behavior
      @     requires true;
      @   assignable \nothing;
      @      ensures \result == 4;
      @*/
    int m() {
        C c = new C();
        c.a = c;
        c.i = 2;
        return c.i + c.a.i;
    }
}
```

Fig. 1. Aliasing via Field References.

In case the field `b` in Figure 2 is true, method `m()` yields $\texttt{f()} \vee \neg\texttt{f()} = \mathsf{false}$ and $\neg\texttt{f()} \wedge \texttt{f()} = \mathsf{true}$, going against standard logical rules.

The verification of the specification for method `m()` may use either the implementation or the specification of `f()`.

3.3 Breaking out of a loop

While and for loops are typically used for going through an enumeration, for instance to find or modify an entry meeting a specific condition. Upon hitting this entry, the loop may be aborted via a break statement. This presents a challenge for the underlying control flow semantics.

Figure 3 presents a simple example of a for loop that goes through an array of integers in order to change the sign of the first negative entry. The two lines of java code are annotated with the loop invariant, with JML-keyword `maintaining` stating what holds while going through the loop, and the loop variant, with JML-keyword `decreasing`. The loop variant is a mapping to the natural numbers which decreases with every loop cycle. It is used in verifications to show that the repetition terminates.

```
boolean b = true; boolean result1, result2;

/*@ normal_behavior
  @     requires true;
  @   assignable b;
  @      ensures b == !\old(b) && \result == b;
  @*/
boolean f() { b = !b; return b; }

/*@ normal_behavior
  @     requires true;
  @   assignable b, result1, result2;
  @      ensures (\old(b) ==> !result1 && result2) &&
  @              (!\old(b) ==> result1 && result2);
  @*/
void m() { result1 = f() || !f(); result2 = !f() && f(); }
```

Fig. 2. Side-effects and Conditional Logical Operators.

The result of ESC/Java on this program is not very interesting because of its limited handling of loops: they are executed (symbolically) only once by default. In general, this may indicate a basic problem with the invariant, but the coverage is far from complete[5].

3.4 Catching exceptions

Typical of Java is its systematic use of exceptions, via its statements for throwing and catching. They require a suitable control flow semantics. Special care is needed for the 'finally' part of a try-catch-finally construction. Figure 4 contains a simple example (adapted from [17]) that combines many aspects. The subtle point is that the assignment `m+=10` in the finally block will still be executed, despite the earlier return statements, but has no effect on the value that is returned. The reason is that this value is bound earlier.

3.5 Bitwise operations

Our next example in Figure 5 is not of the sort one finds in textbooks on program verification. But it is a good example of the ugly code that verification tools have to deal with in practice, specifically in Java Card applets[6]. It involves a "command" byte `cmd` which is split in two parts: the first three, and last five bits. Depending on these parts, a mode field is given an appropriate value. This

[5] As an aside: ESC/Java has difficulty with this example due to limitations in its parser as quantified expressions cannot be used in ternary operations. If we rewrite the specification of `negatefirst` as a conjuction of disjoint implications, ESC/Java accepts the program.

[6] ESC/Java does not handle a bitwise operator like signed right shift (`>>`) correctly.

```
int[] ia;

/*@ normal_behavior
  @     requires ia != null;
  @   assignable ia[*];
  @      ensures \forall int i; 0 <= i && i < ia.length ==>
  @          (\old(ia[i]) < 0 &&
  @          (// i is the first position with negative value
  @           \forall int j; 0 <= j && j < i ==> \old(ia[j]) >= 0))
  @          ? (ia[i] == -\old(ia[i]))
  @          : (ia[i] == \old(ia[i]));
  @*/
void negatefirst() {
    /*@ maintaining i >= 0 && i <= ia.length &&
      @           (\forall int j; 0 <= j && j < i ==>
      @                (ia[j] >= 0 && ia[j] == \old(ia[j])));
      @ decreasing ia.length - i;
      @*/
    for(int i = 0; i < ia.length; i++) {
        if (ia[i] < 0) { ia[i] = -ia[i]; break; } }
}
```

Fig. 3. Breaking out of a Repetition.

```
int m;

/*@ normal_behavior
  @     requires true;
  @   assignable m;
  @      ensures \result == ((d == 0) ? \old(m) : \old(m) / d)
  @              && m == \old(m) + 10;
  @*/
int returnfinally(int d) {
    try { return m / d; }
    catch(Exception e) { return m / (d+1); }
    finally { m += 10; }
}
```

Fig. 4. Return within try-catch-finally.

happens in a nested switch. The specification is helpful because it tells in decimal notation what is going on.

3.6 Class invariants and callbacks

Class invariants are extremely useful in specification, because they often make explicit what programmers have in the back of their mind while writing their

```
static final byte ACTION_ONE = 1, ACTION_TWO = 2,
                  ACTION_THREE = 3, ACTION_FOUR = 4;
private /*@ spec_public @*/ byte mode;

/*@ behavior
  @     requires true;
  @   assignable mode;
  @      ensures (cmd == 0 && mode == ACTION_ONE)    ||
  @              (cmd == 16 && mode == ACTION_TWO)   ||
  @              (cmd == 4 && mode == ACTION_THREE) ||
  @              (cmd == 20 && mode == ACTION_FOUR);
  @      signals (Exception)
  @              ((cmd & 0x07) != 0 || (cmd != 0 && cmd != 16))
  @              &&
  @              ((cmd & 0x07) != 4 || (cmd != 4 && cmd != 20));
  @*/
void selectmode(byte cmd) throws Exception {
    byte cmd1 = (byte)(cmd & 0x07), cmd2 = (byte)(cmd >> 3);
    switch (cmd1) {
    case 0x00: switch (cmd2) {
               case 0x00: mode = ACTION_ONE; break;
               case 0x02: mode = ACTION_TWO; break;
               default: throw new Exception(); }
               break;
    case 0x04: switch (cmd2) {
               case 0x00: mode = ACTION_THREE; break;
               case 0x02: mode = ACTION_FOUR; break;
               default: throw new Exception(); }
               break;
    default: throw new Exception(); }
    // ... more code
}
```

Fig. 5. Typical Mode Selection Based on Command Byte.

code. A typical example is: “integer `i` is always non-zero” (so that one can safely divide by `i`).

The standard semantics for class invariants is: when an invariant holds in the pre-state of a (non-constructor) method, it must also hold in the post-state. Note that this post-state can result from either normal or exceptional termination. An invariant may thus be temporarily broken within a method body, as long as it is re-established at the end. A simple example is method `decrementk` in Figure 6.

Things become more complicated when inside such a method body the class invariant is broken and another method is called. The current object `this` is then left in an inconsistent state. This is especially problematic if control returns at some later stage to the current object. This re-entrance or callback phenomenon is discussed for instance in [25, Sections 5.4 and 5.5]. The commonly adopted solution to this problem is to require that the invariant of `this` is established before a method call. Hence the proof obligation in a method call `a.m()` involves the invariants of both the caller (`this`) and the callee (`a`).

This semantics is incorporated in the translation performed by the LOOP tool. Therefore we can *not* prove the specification for the method `incrementk` in Figure 6. However, a proof using the implementations of method `go` and `decrementk` is possible, if we make the additional assumptions that the run-time type of the field `b` is actually `B`, and that the method `incrementk` is executed on an object of class `A`. These restrictions are needed because if, for instance, field `b` has a subclass of `B` as run-time type, a different implementation will have to be used if the method `go` is overridden in the subclass.

ESC/Java warns about the potential for invariant violation during the call-back.

Another issue related to class invariant is whether or not they should be maintained by *private* methods. JML does require this, but allows a special category of so-called 'helper' methods which need not maintain invariants. We don't discuss this matter further.

3.7 Static initialization

Figure 7 shows an example of static initialization in Java (due to Jan Bergstra). In Java a class is initialized at its first *active* use (see [13]). This means that class initialization in Java is lazy, so that the result of initialization depends on the order in which classes are initialized. The rather sick example in Figure 7 shows what happens when two classes, which are not yet initialized, have static fields referring to each other. In the specification we use a new keyword `\static_fields_of` in the assignable clause. It is syntactic sugar for all static fields of the class.

The first assignment in the body of method `m()` triggers the initialization of class `C1`, which in turn triggers the initialization of class `C2`. The result of the whole initialization is, for instance, that static field `C2.b2` gets value `false` assigned to it. This can be seen when one realizes that the boolean static fields from class `C1` initially get the default value `false`. Subsequently, class `C2` becomes initialized and its fields also get the default value `false`. Now the assignments in class `C2` are carried out: `d2` is set to `true` and `b2` is set to `false`. Note that `d1` is still `false` at this stage. Finally the assignments to fields in class `C1` take place, both resulting in value `true`.

One can see that the order of initializations is important. When the first two assignments in the method body of `m()` are switched, class `C2` will be initialized before class `C1` resulting in all fields getting value `true`.

ESC/Java cannot handle this example as it cannot reason about static initialization. It provides no warnings for potential run-time errors in static initializers or in initializers for static fields.

```
class A {
    private /*@ spec_public @*/ int k, m;
    B b;

    /*@ invariant k + m == 0; @*/

    /*@ normal_behavior
      @     requires true;
      @   assignable k, m;
      @      ensures k == \old(k) - 1 && m == \old(m) + 1;
      @*/
    void decrementk () { k--; m++; }

    /*@ normal_behavior
      @     requires b != null;
      @   assignable k, m;
      @      ensures true;
      @*/
    void incrementk () { k++; b.go(this); m--; }
}

class B {
    /*@ normal_behavior
      @     requires arg != null;
      @   assignable arg.k, arg.m;
      @      ensures arg.k == \old(arg.k) - 1 &&
      @              arg.m == \old(arg.m) + 1;
      @*/
    void go(A arg) { arg.decrementk(); }
}
```

Fig. 6. Callback with Broken Invariant.

3.8 Overriding and dynamic method invocation

The example in Figure 8 is usually attributed to Kim Bruce. It addresses an issue which is often thought of as confusing in programming with languages which support inheritance. The overriding of the method `equal` makes it hard to tell which implementation is called: the one in the subclass or in the superclass.

When a method is overridden, the run-time type of an object decides which implementation is called. This phenomena is also called *late-binding*. In the example three different objects are created, and the question is which `equal` method will be used.

Notice that the `equal` methods in Figure 8 have no specifications. According to the behavioural subtyping semantics used in JML, the `equal` method in the subclass `ColorPoint` should also satisfy the specification of the `equal` method in superclass `Point`. This makes it impossible to prove a precise specification of

```
class C {
  static boolean result1, result2, result3, result4;

    /*@ normal_behavior
      @   requires !\is_initialized(C)   &&
      @            !\is_initialized(C1)  &&
      @            !\is_initialized(C2);
      @ assignable \static_fields_of(C),
      @            \static_fields_of(C1),
      @            \static_fields_of(C2);
      @    ensures result1 && !result2 && result3 && result4;
      @*/
    static void m(){
        result1 = C1.b1; result2 = C2.b2;
        result3 = C1.d1; result4 = C2.d2;
    }
}

class C1 {
    static boolean b1 = C2.d2;
    static boolean d1 = true;
}


class C2 {
    static boolean d2 = true;
    static boolean b2 = C1.d1;
}
```

Fig. 7. Static Initialization.

the `equal` method in class `ColorPoint`. Therefore we proved the specification of method `m()` by using the implementations of the `equal` methods.

3.9 Inheritance

The program in Figure 9 is from [16] and was originally suggested by Joachim van den Berg. On first inspection it looks like the method `test()` will loop forever.

The method `test()` calls method `m()` from class `C`, which calls method `m()` from class `Inheritance`, since 'this' has runtime-type `Inheritance`. Due to the subtype semantics used in JML for inheritance, we cannot write specifications for both of the `m()` methods with which we can reason. Therefore we can only prove the specification of method `test()` by using the method implementations.

```
class Point {

    int equal(Point x) { return 1; }
}

class ColorPoint extends Point {

    int equal(Point x) { return 2; }
}

     int r1,r2,r3,r4,r5,r6,r7,r8,r9;

     /*@ normal_behavior
       @      requires true;
       @    assignable r1, r2, r3, r4, r5, r6, r7, r8, r9;
       @       ensures r1 == 1 && r2 == 1 && r3 == 2 &&
       @               r4 == 2 && r5 == 2 && r6 == 2 &&
       @               r7 == 1 && r8 == 2 && r9 == 2;
       @*/
     void m() {
         Point p1 = new Point();
         Point p2 = new ColorPoint();
         ColorPoint cp = new ColorPoint();
         r1 = p1.equal(p1);r2 = p1.equal(p2);r3 = p2.equal(p1);
         r4 = p2.equal(p2);r5 = cp.equal(p1);r6 = cp.equal(p2);
         r7 = p1.equal(cp);r8 = p2.equal(cp);r9 = cp.equal(cp);
     }
```

Fig. 8. Overriding and dynamic method invocation.

```
class C {
     void m() throws Exception { m(); }
}

class Inheritance extends C {
     void m() throws Exception { throw new Exception(); }

     /*@ exceptional_behavior
       @      requires true;
       @    assignable \nothing;
       @     signals(Exception) true;
       @*/
     void test() throws Exception { super.m(); }
}
```

Fig. 9. Overriding and Dynamic Types.

3.10 Non-termination

The example in Figure 10 (due to Cees-Bart Breunesse) shows a program that does not terminate.

```
class Diverges{

    /*@behavior
      @ requires true;
      @ assignable \nothing;
      @ ensures false;
      @ signals (Exception e) false;
      @ diverges true;
      @*/
    public void m(){
        for (byte b = Byte.MIN_VALUE; b <= Byte.MAX_VALUE; b++)
            ;
    }
}
```

Fig. 10. A Program that does not Terminate.

The specification asserts that the program does not terminate normally or with an exception. The JML keyword `diverges` followed by the predicate `true` indicates that the program fails to terminate. The reader can easily see that this program does not terminate. Since `Byte.MAX_VALUE + 1 = Byte.MIN_VALUE` the guard in the for loop will never fail. Note that in order to verify this program both overflowing and non-termination have to be modeled appropriately. ESC/Java does not handle non-termination.

3.11 Specification

The final example exemplifies two commonplace complications in reasoning about "real world" Java.

Representations of integral types (e.g., `int`, `short`, `byte`, etc.) in Java are finite. A review of annotated programs that use integral types indicates that specifications are often written with infinite numeric models in mind [7]. Programmers seem to think about the issues of overflow (and underflow, in the case of floating point numbers) in program code, but not in specifications.

Additionally, it is often the case that specifications use functional method invocations. Methods which have no side-effects in the program are called called "pure" methods in JML and "queries" in the Eiffel and UML communities[7].

[7] There is still debate in the community about the meaning of "pure". Many Java methods, for example, which claim to have no side-effects, and thus should be pure, actually do modify the state due to caching, lazy evaluation, etc.

```
/*@ normal_behavior
  @      requires true;
  @    assignable \nothing;
  @       ensures \result == ((x >= 0 ||
  @                             x == Integer.MIN_VALUE) ? x : -x);
  @*/
/*@ pure @*/ int iabs(int x) {
    if (x < 0) return -x; else return x; }

/*@ normal_behavior
  @      requires x >= 0 && x <= 2147390966;
  @    assignable \nothing;
  @       ensures iabs(\result) < 46340 &&
  @               \result * \result <= x &&
  @               x < (iabs(\result) + 1) * (iabs(\result) + 1);
  @*/
int isqrt(int x) {
    int count = 0, sum = 1;
     /*@ maintaining 0 <= count &&
       @             count < 46340 &&
       @             count * count <= x &&
       @             sum == (count + 1) * (count + 1);
       @ decreasing  x - count;
       @*/
    while (sum <= x) { count++; sum += 2 * count + 1; }
    return count;
}
```

Fig. 11. Dependent Specifications and Integral Types.

```
/*@ normal_behavior
  @   requires x >= 0;
  @    ensures Math.abs(\result) <= x &&
  @            \result * \result <= x &&
  @            x < (Math.abs(\result) + 1) *
  @                (Math.abs(\result) + 1);
  @*/
```

Fig. 12. JML Specification of Integer Square Root from [20].

The example in Figure 11 highlights both complications, as it uses method invocations in a specification and integral values that can potentially overflow. The method `isqrt()`, which computes an integer square root of its input, is inspired by a specification (see Figure 12) of a similar function included with early JML releases [20].

Note that the `iabs()` method in Figure 11 is used in the specification of `isqrt()` to stipulate that both the negative and the positive square root are an acceptable return value, as all we care about is its magnitude. Actually, our implementation of `isqrt()` only computes the positive integer square root.

The original specification included in early JML releases is provided in Figure 12. This specification uses the `Math.abs()` method instead of our `iabs()` method. We use our own equivalent implementation of square root out of convenience, not necessity.

The definition of JML states that expressions in the requires and ensures clauses are to be interpreted in the semantics of Java. Consequently, a valid implementation of the specification in Figure 12 is permitted to return `Integer.MIN_VALUE` when `x` is 0 (as already noted in [20]).

This surprising situation exists because Java's integral are bounded and because the definition of unary negation in the Java Language Specification is somewhat unexpected.

Because integral types are bounded, expressions in the postcondition, specifically the multiplication operations, can overflow. Additionally, the two implementations of integer absolute value both return a value of `Integer.MIN_VALUE` when passed `Integer.MIN_VALUE`. While this is the documented behavior of `java.lang.Math.abs(int)`, it is often overlooked by programmers because they presume that a function as mathematically uncomplicated as absolute value will produce unsurprising, mathematically correct results for all input.

The absolute value of `Integer.MIN_VALUE` is equal to itself because `Math.abs()` is implemented with Java's unary negation operator "`-`". This operator, defined in Section 15.15.4 of the Java Language Specification, silently overflows when applied to maximal negative integer or long [13][8].

The precondition of `isqrt()` in Figure 11 is explained in Figure 13. We wish to ensure that no operation, either in the implementation of `isqrt()` or in its specification, overflows. The critical situation that causes an overflow is when we attempt to take an integer square root of a very large number. In particular, if we attempt to evaluate the postcondition of `isqrt()` for values of `x` larger than 2,147,390,966 an overflow takes place. The small but critical interval between the precondition's bound 2,147,390,966 and `Integer.MAX_VALUE` = 2,147,483,647 is indicated by the dark interval on the right of Figure 13: to check the postcondition, the prospective root (the arrow labeled 1) must be determined, one is added to its value (arrow 2), and the result is squared (arrow 3). This final result will thus overflow. Indeed, $(46,340+1)^2 > 2,147,483,647$.

The erroneous nature of a specification involving potential overflows should become clear when one verifies the method using an appropriate bit-level representation of integral types [18]. Unfortunately, such errors are not at all apparent, even when performing extensive unit testing, because the boundary conditions for arithmetic expressions, like the third term of the postcondition of `isqrt()` in Figure 11, are rarely automatically derivable, and full state-space coverage is simply too computationally expensive.

[8] Interestingly, the documentation for `java.lang.Math.abs()` did not reflect this fact until the 1.1 release of Java.

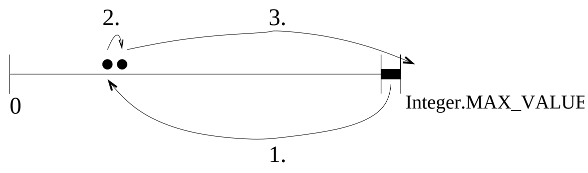

Fig. 13. The Positive Integers.

Specifications involving integral types, and thus potential overflows, are frequently seen in application domains that involve numeric computation both complex (e.g., scientific computation, computer graphics, embedded devices, etc.) and relatively simple (e.g., currency and banking). The former category are obviously challenging due to the complexity of the related data-structures, algorithms, and their specifications, and the latter are problematic because it is there that implementation violations have egregious (financial) consequences. This specification raises the question: What is the appropriate model for arithmetic specifications[9]?

Another challenge highlighted by this example might be called "formalism completeness". Many semantic formalisms of modern programming languages like Java do not attempt to specify the semantics of complicated features like method invocation. Even fewer attempt to incorporate such semantics in method specifications, as used here.

For example, ESC/Java is unable to deal with this example because of such interdependencies. This is a significant limitation as many, if not most, method specifications rely upon pure or JML helper methods. This is also a weakness of the LOOP tool, as dealing with (pure) method calls in specifications is tedious in verifications[10]. In general, the semantics of method invocation in specifications is still an unclear issue at this time.

4 Conclusions

The starting point of this paper is the observation that the classical examples in (sequential) program verification are no longer very relevant in today's context. They need to be updated in two dimensions: language complexity and size. This paper focuses on complexity, by presenting a new series of challenges, written in Java, with correctness assertions expressed in JML. The examples incorporate some of the ugly details that one encounters in real-world programs, and that any reasonable semantics should be able to handle.

The fact that these example programs are small does not mean that we think size is unimportant. On the contrary, once a reasonably broad semantic spectrum

[9] We do not have answers for these questions, though investigations are underway [8].

[10] The verification of the method `isqrt()` from Figure 11 uses the implementation of the absolute value method `iabs()`.

is covered, the next challenge is to scale up one's program verification techniques to larger programs. With our tools we are currently verifying programs with hundreds of lines of code.

Acknowledgments

We want to thank Patrice Chalin for insightful comments on an earlier version of this paper.

References

1. Erika Ábrahám Mumm, Frank S. de Boer, Willem-Paul de Roever, and Martin Steffen. A Tool-supported Proof System for Multithreaded Java. In *FMCO 2002 proceedings*, Lect. Notes Comp. Sci. Springer, Berlin, 2003.
2. W. Ahrendt, Th. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P.H. Schmitt. The KeY tool. Technical Report 2003-5, Chalmers and Göteborg University, 2003. `http://i12www.ira.uka.de/~key`.
3. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001.
4. J. Bergstra and M. Loots. Empirical semantics for object-oriented programs. Artificial Intelligence Preprint Series nr. 007, Dep. Philosophy, Utrecht Univ. `http://preprints.phil.uu.nl/aips/`, 1999.
5. E. Börger and W. Schulte. Initialization problems in Java. *Software—Concepts and Tools*, 20(4), 1999.
6. Lilian Burdy, Jean-Louis Lanet, and Antoine Requet. JACK (Java Applet Correctness Kit), 2002. `www.gemplus.com/smart/r_d/trends/jack.html`.
7. P. Chalin. Back to basics: Language support and semantics of basic infinite integer types in JML and Larch. Technical Report 2002.003.1, Computer Science Department, Concordia University, 2002. `www.cs.concordia.ca/~faculty/chalin/`.
8. P. Chalin. Improving JML: For a safer and more effective language. Technical Report 2003-001.1, Computer Science Department, Concordia University, March 2003.
9. Y. Cheon and G.T. Leavens. A runtime assertion checker for the Java Modeling Language (JML). In H.R. Arabnia and Y. Mun, editors, *International Conference on Software Engineering Research and Practice (SERP '02)*, pages 322–328. CSREA Press, Las Vegas, 2002.
10. E. Contejean, J. Duprat, J.-C. Filliâtre, C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for JML/Java program certification. Available via the Krakatoa home page at `www.lri.fr/~marche/krakatoa/`, October 2002.
11. E.W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
12. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37(5) of *SIGPLAN Notices*, pages 234–245. ACM, 2002.

13. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition.* The Java Series. Addison-Wesley, 2000. `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`.
14. D. Gries. *The Science of Programming.* Springer, 1981.
15. C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, 583, 1969.
16. M. Huisman and B. Jacobs. Inheritance in higher order logic: Modeling and reasoning. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lect. Notes Comp. Sci., pages 301–319. Springer, Berlin, 2000.
17. B. Jacobs. A formalisation of Java's exception mechanism. In D. Sands, editor, *Programming Languages and Systems (ESOP)*, number 2028 in Lect. Notes Comp. Sci., pages 284–301. Springer, Berlin, 2001.
18. B. Jacobs. Java's integral types in PVS. Manuscript, `http://www.cs.kun.nl/~bart/PAPERS/`, 2003.
19. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov and B. Rumpe, editors, *Behavioral Specifications of Business and Systems*, pages 175–188. Kluwer, 1999.
20. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. `http://www.cs.iastate.edu/~leavens/JML.html`, 1999.
21. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual (draft). `www.jmlspecs.org`, 2002.
22. B. Liskov and J.M. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. & Syst.*, 7(16(6)):1811–1841, 1994.
23. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 1785 in Lect. Notes Comp. Sci., pages 63–77. Springer, Berlin, 2000.
24. E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Smart Card Research and Advanced Application*, pages 135–154. Kluwer Acad. Publ., 2000.
25. C. Szyperski. *Component Software.* Addison-Wesley, 1998.