Java's Integral Types in PVS ⋆

Bart Jacobs

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

Abstract. This paper presents an extension of the standard bitvector library of the theorem prover PVS with multiplication, division and remainder operations, together with associated results. This extension is needed to give correct semantics to Java's integral types in program verification. Special emphasis is put on Java's widening and narrowing functions in relation to the newly defined operations on bitvectors.

1 Introduction

Many programmming languages offer different integral types, represented by different numbers of bits. In Java, for instance, one has integral types `byte` (8 bits), `short` (16 bits), `int` (32 bits) and `long` (64 bits). Additionally, there is a 16 bit type `char` for unicode characters, see [8, §§4.2.1]. It is a usual abstraction in program verification to disregard these differences and interpret all of these types as the unbounded, mathematical integers. However, during the last few years both program verification and theorem proving technology have matured in such a way that more precise representations of integral types can be used.

An important application area for program verification is Java Card based smart cards. Within this setting the above mentioned abstraction of integral types is particularly problematic, because of the following reasons.

– Given the limited resources on a smart card, a programmer chooses his/her integral data types as small as possible, so that potential overflows are a concern (see [6, Chapter 14]). Since such overflows do not produce exceptions in Java (like in Ada), a precise semantics is needed.
– Communication between a smart card and a terminal uses a special structured byte sequence, called an "apdu", see [6]. As a result, many low-level operations with bytes occur frequently, such as bitwise negation or masking.
– Unnoticed overflow may form a security risk: imagine you use a short for a sequence number in a security protocol, which is incremented with every protocol run. An overflow then makes you vulnerable to a possible replay attack.

Attention in the theorem proving community has focused mainly on formalising properties of (IEEE 754) floating-point numbers, see *e.g.* [4, 9, 10, 18]. Such results are of interest in the worlds of microprocessor construction and scientific computation.

⋆ To appear in: LNCS proceedings of FMOODS 2003.

However, there are legitimate concerns about integral types as well. It is argued in [19] that Java's integral types are unsafe, because overflow is not detected via exceptions, and are confusing because of the asymmetric way that conversions work: arguments are automatically promoted, but results are not automatically "demoted"[1].

Verification tools like LOOP [3] or Krakatoa [14] use the specification language JML [12, 13] in order to express the required correctness properties for Java programs. Similarly, the KeY tool [1] uses UML's Object Constraint Language (OCL). Program properties can also be checked statically with the ESC/Java tool [7], but such checking ignores integral bounds. The theorem prover based approach with the semantics of this paper will take bounds into account. In [5] (see also [2]) it is proposed that a specification language like JML for Java should use the mathematical (unbounded) integers, for describing the results of programs using bounded integral types, because "developers are in a different mindset when reading or writing specifications, particularly when it comes to reasoning about integer arithmetic". This issue is not resolved yet in the program specification community—and it will not be settled here.

Instead, this paper describes the bit-level semantics for Java's integral types developed for the LOOP tool. As such it contributes both to program verification and to library development for theorem provers (esp. for PVS [15]). The semantics is based on PVS's (standard) bitvector library. This PVS library describes bitvectors of arbitrary length, given as a parameter, together with functions bv2nat and bv2int for the unsigned (one's-complement) and signed (two's-complement) interpretation of bitvectors. Associated basic operations are defined, such as addition, subtraction, and concatenation. In this paper, the following items are added to this library.

1. Executable definitions. For instance, the standard library contains "definitions by specification" of the form:

   ```
   -(bv: bvec[N]): { bvn: bvec[N] | bv2int(bvn) =
                         IF bv2int(bv) = minint THEN bv2int(bv)
                         ELSE -(bv2int(bv)) ENDIF}
   *(bv1: bvec[N], bv2: bvec[N]): {bv: bvec[2*N] | bv2nat(bv) =
                                      bv2nat(bv1) * bv2nat(bv2)}
   ```

   Such definitions[2] are not so useful for our program verifications, because sometimes we need to actually compute outcomes. Therefore we give executable redefinitions of these operations. Then we can compute, for instance, `(4*b)&0x0F`.
2. Similarly, executable definitions are introduced for division and remainder operations, which are not present in the standard library. We give such definitions both for unsigned and signed interpretations, following standard hardware realisations via shifting of registers. The associated results are non-trivial challenges in theorem proving.

[1] For a byte (or short) `b`, the assignment `b = b-b` leads to a compile time error: the arguments of the minus function are first converted implicitly to `int`, but the result must be converted explicitly back, as in `b = (byte)(b-b)`.

[2] Readers familiar with PVS will see that these definitions generate so-called type correctness conditions (TCCs), requiring that the above sets are non-empty. These TCCs can be proved via the inverses `int2bv` and `nat2bv` of the (bijective) functions `bv2int` and `bv2nat`, see Section 3. The inverses exist because one has bijections, but they are not executable.

3. Specifically for Java we introduce so-called widening and narrowing, for turning a bitvector of length N into one of length $2 * N$ and back, see [8, §§5.1.2 and §§5.1.3]. When, for example, a byte is added to a short, both arguments are first "promoted" in Java-speak to integers via widening, and then added. Appropriate results are proven relating for instance widening and multiplication or division.

We show how our definitions of multiplication, division and remainder satisfy all properties listed in the Java Language Specification [8, §§15.17.1-3].

In particular we get a good handle on overflow, so that we can prove for the values `minint = 0x80000000` and `maxint = 0x7FFFFFFF`, the truth of the following Java boolean expressions.

```
minint - 1 == maxint       maxint + 1 == minint
minint * -1 == minint      maxint * maxint == 1
minint / -1 == minint
```

As a result the familiar cancellation laws for multiplication ($a * b = a * c \Rightarrow b = c$, for $a \neq 0$) and for division ($\frac{a*b}{a*c} = \frac{b}{c}$, for $a \neq 0, c \neq 0$) do not hold, since:

```
minint * -1 == minint * 1    (minint * -1) / (minint * 1) == 1
```

But these cancellation laws do hold in case there is no overflow. Similarly, we can prove the crucial property of a mask to turn bytes into nonnegative shorts: for a byte `b`,

```
(short)(b & 0xFF) == (b >= 0) ? b : (b + 256)
```

Two more examples are presented in Section 2.

Integral arithmetic is a very basic topic in computer science (see *e.g.* [17]). Most theorem provers have a standard bitvector library that covers the basics, developed mostly for hardware verification. But multiplication, division and remainder are typically not included. The contribution of this paper lies in the logical formalisation of these operations and their results, and in linking the outcome to Java's arithmetic, especially to its widening and narrowing operations. These are the kind of results that "everybody knows" but are hard to find and easy to get wrong.

Of course, one can ask: why go through all this trouble at bitvector level, and why not define the integral operations directly on appropriate bounded intervals of the (mathematical) integers—like for instance in [16] (for the theorem prover Isabelle)? We have several reasons.

– Starting at the lowest level of bits gives more assurance. The non-trivial definitions of the operations that should be used on the bounded intervals appear in our bit-level approach as results about operations that are defined at a lower level (see the final results before Subsection 7.1). This is important, because for instance in [16] it took several iterations (with input from the present approach) to get the correct formulation for the arithmetically non-standard definitions of division and remainer for Java.
– Once appropriate higher-level results are obtained about the bit-level representation, these results can be used for automatic rewriting, without revealing the underlying structure. Hence this approach is at least as powerful as the one with bounded intervals of integers.

– Certain operations from programming languages (like bitwise conjunction or negation) are extremely hard to describe without an underlying bit-level semantics.

This paper has a simple structure. It starts by describing simple Java programs with integral types as motivation. From there it investigates bitvectors. First it explains the basics of PVS's standard bitvector library. Then, in Section 5 it describes our definition of multiplication with associated properties. Division and remainder operations are more difficult; they are first described in unsigned (one's-complement) form in Section 6, and subsequently in signed (two's-complement) form in Section 7. Although the work we have done has been carried out in the language of a specific theorem prover (namely PVS), we shall use a general, mathematical notation to describe it.

2 Java examples

The following two programs[3] are extremely simple, yet involve some non-trivial semantical issues.

```
int n() {
  for (byte b = Byte.MIN_VALUE;
       b <= Byte.MAX_VALUE; b++) {
         if (b == 0x90) {
               return 1000; };
  }
}
```

```
int s() {
  int n = 0;
  while (-1 << n != 0) {
     n++;
  };
  return n;
}
```

Both these programs hang (*i.e.* loop forever). This can be shown with the LOOP tool, using the integral semantics described in this paper. The reader may wish to pause a moment to understand why these programs hang.

The program `n` on the left hangs because the bound condition `b <= Byte.MAX_VALUE` is always true: the increment operation wraps around. Further, the value `0x90` is interpreted in Java as an integer, and is thus equal to $9 \times 16 = 144$, which is outside the range $[-128, 127]$ used for bytes. Hence the condition of the if-statement is always false, so that one does not return from the loop in this way.

Within the program `s` on the right the integral -1 (which is `0xFFFFFFFF`) is repeatedly shifted to the left. However, Java's leftshift operator << only uses the five lower-order bits of its second argument, see [8, §§15.19]. These five bits can result in a shift of at most $s^5 - 1 = 31$ positions. This is not enough to turn -1 into 0.

This illustrates that a proper understanding of ranges and bitpositions is needed to reason about even elementary Java programs.

3 PVS's standard bitvector library

The distribution of PVS comes with a basic bitvector library[4]. We sketch some ingredients that will be used later. A bit is defined as in PVS as a boolean, but here

[3] Adapted from `www.linux-mag.com/downloads/2003-03/puzzlers/`

[4] Developed by Butler, Miner, Carreño (NASA Langley), Miller, Greve (Rockwell Collins) and Srivas (SRI International).

we shall equivalently use it as an element of $\{0,1\}$. A bitvector of length N is a function in $\mathsf{bvec}(N) \stackrel{\text{def}}{=} \big(\mathsf{below}(N) \to \mathsf{bit}\big)$, where $\mathsf{below}(N)$ is the N-element set $\{0,1,\ldots,N-1\}$ of natural numbers below N. For instance, the null bitvector is $\lambda i \in \mathsf{below}(N).\,0$, which we shall often write as $\overrightarrow{0}$, leaving the length N implicit. Similarly, one can write $\overrightarrow{1}$ for $\lambda i \in \mathsf{below}(N).\,1$. It should be distinguished from $\mathbf{1} = \lambda i \in \mathsf{below}(N).\,\mathsf{if}\ i = 0\ \mathsf{then}\ 1\ \mathsf{else}\ 0$.

The unsigned interpretation of bitvectors is given by the (parametrised) function $\mathsf{bv2nat}\colon \mathsf{bvec}(N) \to \mathsf{below}(2^N)$, defined as:

$$\begin{aligned} \mathsf{bv2nat}(bv) &\stackrel{\text{def}}{=} \mathsf{bv2nat\text{-}rec}(bv, N), \quad \text{where} \\ \mathsf{bv2nat\text{-}rec}(bv, n) &\stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } n = 0 \\ bv(n-1) * 2^{n-1} + \mathsf{bv2nat\text{-}rec}(bv, n-1) & \text{if } n > 0 \end{cases} \end{aligned} \tag{1}$$

Clearly, $\mathsf{bv2nat}$ is bijective. And also: $\mathsf{bv2nat}(bv) = 0 \Leftrightarrow bv = \overrightarrow{0}$, $\mathsf{bv2nat}(bv) = 2^N - 1 \Leftrightarrow bv = \overrightarrow{1}$, and $\mathsf{bv2nat}(bv) = 1 \Leftrightarrow bv = \mathbf{1}$.

The signed interpretation is given by a similar function $\mathsf{bv2int}\colon \mathsf{bvec}(N) \to \{i \in \mathbb{Z} \mid -2^{N-1} \le i \wedge i < 2^{N-1}\}$. It is defined in terms of the unsigned interpretation:

$$\mathsf{bv2int}(bv) \stackrel{\text{def}}{=} \begin{cases} \mathsf{bv2nat}(bv) & \text{if } \mathsf{bv2nat}(bv) < 2^{N-1} \\ \mathsf{bv2nat}(bv) - 2^N & \text{otherwise.} \end{cases} \tag{2}$$

The condition $\mathsf{bv2nat}(bv) < 2^{N-1}$ means that the most significant bit $bv(N-1)$ is 0. Therefore, this bit is often called the *sign bit*, when the signed interpretation is used. This $\mathsf{bv2int}$ function is also bijective.

The PVS bitvector library provides various basic operations and results. For instance, there is an (executable) addition operation $+$ on bitvectors, introduced via a recursively defined adder. A unary minus operation $-$ is introduced via a specification, as described in the introduction. Binary minus is then defined as: $bv_1 - bv_2 \stackrel{\text{def}}{=} bv_1 + (-bv_2)$. These operations work for both the unsigned and for the signed interpretation. A typical result is:

$$\begin{aligned} &\mathsf{bv2int}(bv_1 + bv_2) \\ &= \begin{cases} \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) & \text{if } -2^{N-1} \le \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) \\ & \text{and } \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) < 2^{N-1} \\ \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) - 2^N & \text{if } \mathsf{bv2int}(bv_1) \ge 0 \text{ and } \mathsf{bv2int}(bv_2) \ge 0 \\ \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) + 2^N & \text{otherwise.} \end{cases} \end{aligned}$$

The second cases deals with overflow, and the third one with underflow. The library shows, among other things, that the structure $(\mathsf{bvec}(N), +, \overrightarrow{0}, -)$ is a commutative group.

Also we shall make frequent use of left and right shift operations. For $k \in \mathbb{N}$,

$$\begin{aligned} \mathsf{lsh}(k, bv) &= \lambda i \in \mathsf{below}(N).\, \begin{cases} bv(i-k) & \text{if } i \geq k \\ 0 & \text{otherwise.} \end{cases} \\ \mathsf{rsh}(k, bv) &= \lambda i \in \mathsf{below}(N).\, \begin{cases} bv(i+k) & \text{if } i+k < N \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

4 Widening and narrowing

As mentioned in the introduction, Java uses so-called widening and narrowing operations to move from one integral type to another. These operations can be described in a parametrised way, as functions:

$$\mathsf{widen}\colon \mathsf{bvec}(N) \to \mathsf{bvec}(2*N) \quad \text{and} \quad \mathsf{narrow}\colon \mathsf{bvec}(2*N) \to \mathsf{bvec}(N)$$

defined as:

$$\begin{aligned} \mathsf{widen}(bv) &\stackrel{\text{def}}{=} \lambda i \in \mathsf{below}(2*N).\, \begin{cases} bv(i) & \text{if } i < N \\ bv(N-1) & \text{otherwise} \end{cases} \\ \mathsf{narrow}(BV) &\stackrel{\text{def}}{=} \lambda i \in \mathsf{below}(N).\, BV(i) \end{aligned}$$

Thus, narrowing simply ignores the first N bits. The key property of widening is that the unsigned interpretation is unaffected, in the sense that:

$$\mathsf{bv2int}(\mathsf{widen}(bv)) = \mathsf{bv2int}(bv)$$

A theme that will re-appear several times in this paper is that after widening there is no overflow:

$$\begin{aligned} \mathsf{bv2int}\big(\mathsf{widen}(bv_1) + \mathsf{widen}(bv_2)\big) &= \mathsf{bv2int}(bv_1) + \mathsf{bv2int}(bv_2) \\ \mathsf{bv2int}\big(-\mathsf{widen}(bv)\big) &= -\mathsf{bv2int}(bv). \end{aligned} \tag{3}$$

There are similar results about narrowing. First:

$$\mathsf{narrow}(\mathsf{widen}(bv)) = bv.$$

But also:

$$\begin{aligned} \mathsf{narrow}(BV_1 + BV_2) &= \mathsf{narrow}(BV_1) + \mathsf{narrow}(BV_2) \\ \mathsf{narrow}(-BV) &= -\mathsf{narrow}(BV). \end{aligned}$$

The LOOP tool uses widening and narrowing in the translation of Java's arithmetical expressions. For instance, for a byte `b` and short `s`, a Java expression

```
(short)(b + 2*s)
```

is translated into PVS as:

$$\mathsf{narrow}\Big(\mathsf{widen}(\mathsf{widen}(b)) + 2 * \mathsf{widen}(s)\Big)$$

because the arguments are "promoted" in Java to 32 bit integers before addition and multiplication are applied.

In this way we can explain (and verify in PVS) that for `byte b = -128`, one has in Java: `b-1` is -129 and `(byte)(b-1)` is 127.

5 Multiplication

This section describes bitvector multiplication in PVS, following the standard pen-and-paper approach via repeated shifting and adding. The definition we use works well under both the unsigned and signed interpretation.

In our parametrised setting, we use a recursive definition for multiplication. For two bitvectors $bv_1, bv_2 \colon \mathsf{bvec}(N)$ of length N, we define:

$$bv_1 * bv_2 \stackrel{\text{def}}{=} \mathsf{times\text{-}rec}(bv_1, bv_2, N)$$

where for a natural number n,

$$\begin{aligned}
&\mathsf{times\text{-}rec}(bv_1, bv_2, n)\\
&\stackrel{\text{def}}{=} \begin{cases} \overrightarrow{0} & \text{if } n = 0 \\ bv_2 + \mathsf{lsh}(1, \mathsf{times\text{-}rec}(\mathsf{rsh}(1, bv_1), bv_2, n-1)) & \text{if } n > 0 \text{ and } bv_1(0) = 1 \\ \mathsf{lsh}(1, \mathsf{times\text{-}rec}(\mathsf{rsh}(1, bv_1), bv_2, n-1)) & \text{if } n > 0 \text{ and } bv_1(0) = 0 \end{cases}
\end{aligned}$$

Note that in this definition $bv_1 * bv_2$ has the same length as bv_1 and bv_2, unlike the multiplication by specification from the standard PVS library (described in the introduction), which doubles the length.

A crucial result is that (our) multiplication can be expressed simply as iterated addition, an appropriate number of times.

$$bv_1 * bv_2 = \mathsf{iterate}(\lambda b \in \mathsf{bvec}(N).\, b + bv_1, \mathsf{bv2nat}(bv_2))(\overrightarrow{0}),$$

where $\mathsf{iterate}(f, k)(x)$ is $f^{(k)}(x) = f(\cdots f(x) \cdots)$, *i.e.* f applied k times to x. This allows us to prove familiar results, like

$$\begin{gathered}
\overrightarrow{0} * bv = \overrightarrow{0} \quad bv_1 * bv_2 = bv_2 * bv_1 \quad \mathbf{1} * bv = bv \quad (-bv_1) * bv_2 = -(bv_1 * bv_2) \\
bv_1 * (bv_2 * bv_3) = (bv_1 * bv_2) * bv_3 \qquad bv_1 * (bv_2 + bv_3) = bv_1 * bv_2 + bv_1 * bv_3
\end{gathered}$$

They express that $(\mathsf{bvec}(N), *, \mathbf{1})$ is a commutative monoid, and that $*$ preserves the group structure $(\mathsf{bvec}(N), +, \overrightarrow{0}, -)$.

As for the interpretation, the following two results are most relevant.

$$\begin{aligned}
&\mathsf{bv2nat}(bv_1) * \mathsf{bv2nat}(bv_2) < 2^N \\
&\qquad \Longrightarrow \mathsf{bv2nat}(bv_1 * bv_2) = \mathsf{bv2nat}(bv_1) * \mathsf{bv2nat}(bv_2) \\
&-2^{N-1} < \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) \;\text{ and }\; \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) < 2^{N-1} \\
&\qquad \Longrightarrow \mathsf{bv2int}(bv_1 * bv_2) = \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2).
\end{aligned}$$

This means that we have an analogue of (3) for multiplication: after widening there is no overflow:

$$\mathsf{bv2int}(\mathsf{widen}(bv_1) * \mathsf{widen}(bv_2)) = \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2). \tag{4}$$

This result is very useful in actual calculations (in PVS, about Java programs). For the general situation, with possible over- or under-flow, we have the following formula that slices the (mathematical) integers into appropriate ranges.

$$\begin{aligned}\forall n \in \mathbb{Z}.\ & n * 2^N - 2^{N-1} \leq \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) \ \wedge \\ & \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) < n * 2^N + 2^{N-1} \\ & \qquad \Longrightarrow \mathsf{bv2int}(bv_1 * bv_2) = \mathsf{bv2int}(bv_1) * \mathsf{bv2int}(bv_2) - n * 2^N.\end{aligned}$$

Finally, we have the following two results about multiplication and narrowing.

$$\begin{aligned}\mathsf{narrow}(BV_1 * BV_2) &= \mathsf{narrow}(BV_1) * \mathsf{narrow}(BV_2) \\ bv_1 * bv_2 &= \mathsf{narrow}\Big(\mathsf{widen}(bv_1) * \mathsf{widen}(bv_2)\Big)\end{aligned} \tag{5}$$

This second result follows immediately from the first. It is of interest because it shows that our multiplication satisfies the following requirement from the Java Language Specification [8, §§15.17.1]:

> If an integer multiplication overflows, then the result is the low-order bits of the mathematical product as represented in some sufficiently large two's-complement format.

The "lower-order bits" result from the narrow in $\mathsf{narrow}\big(\mathsf{widen}(bv_1) * \mathsf{widen}(bv_2)\big)$ in (5)'s second equation, and the "mathematical product" is its argument $\mathsf{widen}(bv_1) * \mathsf{widen}(bv_2)$, as expressed by (4).

6 Unsigned division and remainder

Division and remainder for bitvectors are less straightforward than multiplication. They are based on the same pen-and-paper principles, but the verifications are more involved. In this section we describe a standard machine algorithm for the unsigned interpretation, see *e.g.* [17, 8.3]. The next section adapts this approach to the signed interpretation, and shows how it can be used for division in Java.

In the description below we use arbitrary bitvectors $dvd, dvs, rem, quot, aux$, of a fixed length N. The abbreviation *dvd* stands for 'dividend', and *dvs* for 'divisor', to be used in $dvd \ / \ dvs$ and $dvd \ \% \ dvs$.

Unsigned division and remainder are defined via first and second projections of a recursive auxiliary function:

$$\begin{aligned}\mathsf{div}(dvd, dvs) &\stackrel{\text{def}}{=} \pi_1 \mathsf{divrem}(dvd, dvs, \overrightarrow{0}, N) \\ \mathsf{rem}(dvd, dvs) &\stackrel{\text{def}}{=} \pi_2 \mathsf{divrem}(dvd, dvs, \overrightarrow{0}, N),\end{aligned}$$

where for $n \in \mathbb{N}$,

$$\begin{array}{rl}
\mathsf{divrem}(\mathit{dvd}, \mathit{dvs}, \mathit{rem}, 0) \stackrel{\text{def}}{=} & (\mathit{dvd}, \mathit{rem}) \\
\mathsf{divrem}(\mathit{dvd}, \mathit{dvs}, \mathit{rem}, n+1) \stackrel{\text{def}}{=} & \mathsf{let}\ \ \mathit{dvd}' = \mathsf{lsh}(1, \mathit{dvd}), \\
& \qquad \mathit{rem}' = \mathsf{lsh}(1, \mathit{rem})\ \mathsf{with}\ [(0) := \mathit{dvd}(N-1)] \\
& \ \ \mathsf{in}\ \ \mathsf{if}\ \mathsf{bv2nat}(\mathit{dvs}) \leq \mathsf{bv2nat}(\mathit{rem}') \\
& \qquad \mathsf{then}\ \mathsf{divrem}(\mathit{dvd}'\ \mathsf{with}\ [(0) := 1], \\
& \qquad\qquad\qquad \mathit{dvs}, \mathit{rem}' - \mathit{dvs}, n-1) \\
& \qquad \mathsf{else}\ \mathsf{divrem}(\mathit{dvd}', \mathit{dvs}, \mathit{rem}', n-1).
\end{array}$$

The 'with' operator is a convenient short-hand for function update: $f\ \mathsf{with}\ [(i) := a]$ is the function g with $g(n) = \mathsf{if}\ n = i\ \mathsf{then}\ a\ \mathsf{else}\ f(n)$.

The definition of divrem is quite sneaky and efficient, since it uses very few arguments (or registers in a hardware implementation). The *dvd* argument is shifted from the left into *rem*, and at the same time the quotient result is built up in *dvd* from the right. This overloaded use of *dvd* makes it impossible to formulate an appropriate invariant for this recursive function. Therefore we introduce an alternative function divrem^* without this overloading, show that divrem^* computes the same result as divrem, and formulate and prove an appropriate invariant for divrem^*.

$$\begin{array}{l}
\mathsf{divrem}^*(\mathit{aux}, \mathit{dvd}, \mathit{dvs}, \mathit{rem}, 0) \stackrel{\text{def}}{=} \\
\qquad (\mathit{aux}, \mathit{dvd}, \mathit{rem}) \\
\mathsf{divrem}^*(\mathit{aux}, \mathit{dvd}, \mathit{dvs}, \mathit{rem}, n+1) \stackrel{\text{def}}{=} \\
\qquad \mathsf{let}\ \ \mathit{aux}' = \mathsf{lsh}(1, \mathit{aux})\ \mathsf{with}\ [(0) := \mathit{dvd}(N-1)] \\
\qquad\qquad \mathit{dvd}' = \mathsf{lsh}(1, \mathit{dvd}), \\
\qquad\qquad \mathit{rem}' = \mathsf{lsh}(1, \mathit{rem})\ \mathsf{with}\ [(0) := \mathit{dvd}(N-1)] \\
\qquad \ \ \mathsf{in}\ \ \mathsf{if}\ \mathsf{bv2nat}(\mathit{dvs}) \leq \mathsf{bv2nat}(\mathit{rem}') \\
\qquad\qquad \mathsf{then}\ \mathsf{divrem}^*(\mathit{aux}', \mathit{dvd}'\ \mathsf{with}\ [(0) := 1], \mathit{dvs}, \\
\qquad\qquad\qquad\qquad \left(\begin{array}{l} \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < N - n + 1 \\ \qquad\qquad\qquad\qquad \mathsf{then}\ \mathit{rem}'(i)\ \mathsf{else}\ 0 \end{array}\right) - \mathit{dvs}, n-1) \\
\qquad\qquad \mathsf{else}\ \mathsf{divrem}^*(\mathit{aux}', \mathit{dvd}', \mathit{dvs}, \mathit{rem}', n-1).
\end{array}$$

In this definition the original argument *dvd* is also shifted into the *aux* register, so that it is not lost and can be used for the formulation of the invariant. Also, the subtraction of *dvs* from rem', if possible, happens only from the relevant, lower part of rem'.

The fact that divrem and divrem^* compute the same results is expressed as follows. For all $n \leq N$,

$$\begin{array}{rl}
\mathsf{let} & \mathit{dr}^* = \mathsf{divrem}^*(\mathit{aux}, \mathit{dvd}, \mathit{dvs}, \mathit{rem}, n), \\
& \mathit{dr} = \mathsf{divrem}(\mathit{dvd}, \mathit{dvs}, \left(\begin{array}{l} \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < n \\ \qquad\qquad\qquad \mathsf{then}\ \mathit{rem}(i)\ \mathsf{else}\ 0 \end{array}\right), n) \\
\mathsf{in} & \pi_1 \mathit{dr}^* = \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < n\ \mathsf{then}\ \mathit{dvd}(N - n + i)\ \mathsf{else}\ \mathit{aux}(i - n)\ \wedge \\
& \pi_2 \mathit{dr}^* = \pi_1 \mathit{dr}\ \wedge \\
& \pi_3 \mathit{dr}^* = \pi_2 \mathit{dr}.
\end{array}$$

Note that for $n = N$, the third argument of divrem is simply *rem*, and $\pi_1\mathsf{dr}^\star = \mathit{dvd}$. As a result, division and remainder can also be expressed in terms of $\mathsf{divrem}^\star$:

$$\begin{aligned}\mathsf{div}(\mathit{dvd}, \mathit{dvs}) &= \pi_2\mathsf{divrem}^\star(\overrightarrow{0}, \mathit{dvd}, \mathit{dvs}, \overrightarrow{0}, N)\\ \mathsf{rem}(\mathit{dvd}, \mathit{dvs}) &= \pi_3\mathsf{divrem}^\star(\overrightarrow{0}, \mathit{dvd}, \mathit{dvs}, \overrightarrow{0}, N).\end{aligned}$$

We are now in a position to express the key invariant property. For $n \leq N$,

$$\begin{array}{ll}\mathsf{let} & \mathit{dvd\_n} = \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < n\ \mathsf{then}\ \mathit{dvd}(N - n + i)\ \mathsf{else}\ \mathit{aux}(i - n),\\ & \mathit{quot\_n} = \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < N - n\ \mathsf{then}\ \mathit{dvd}(i)\ \mathsf{else}\ 0,\\ & \mathit{rem\_n} = \lambda i \in \mathsf{below}(N).\ \mathsf{if}\ i < n\ \mathsf{then}\ \mathit{dvd}(N - n + i)\ \mathsf{else}\ \mathit{rem}(i - n),\\ & \mathit{dr}* = \mathsf{divrem}^\star(\mathit{aux}, \mathit{dvd}, \mathit{dvs}, \mathit{rem}, n)\\ \mathsf{in} & \mathsf{bv2nat}(\mathit{dvd\_n}) = 2^n * \mathsf{bv2nat}(\mathit{dvs}) * \mathsf{bv2nat}(\mathit{quot\_n}) + \mathsf{bv2nat}(\mathit{rem\_n})\\ & \quad\Longrightarrow\\ & \mathsf{bv2nat}(\pi_1\mathit{dr}*) = \mathsf{bv2nat}(\mathit{dvs}) * \mathsf{bv2nat}(\pi_2\mathit{dr}*) + \mathsf{bv2nat}(\pi_3\mathit{dr}*)\end{array}$$

The proof of this property is far from trivial. The most interesting case is when $n = N$. We then have $\mathit{dvd\_n} = \mathit{dvd}$, $\mathit{quot\_n} = \overrightarrow{0}$ and $\mathit{rem\_n} = \mathit{dvd}$, so that the antecedent of the implication $\Longrightarrow$ trivially holds. This yields a first success:

$$\mathsf{bv2nat}(\mathit{dvd}) = \mathsf{bv2nat}(\mathit{dvs}) * \mathsf{bv2nat}(\mathsf{div}(\mathit{dvd}, \mathit{dvs})) + \mathsf{bv2nat}(\mathsf{rem}(\mathit{dvd}, \mathit{dvs})). \tag{6}$$

It is not hard to prove the expected upperbound for remainder:

$$\mathsf{bv2nat}(\mathit{dvs}) \neq 0 \Longrightarrow \mathsf{bv2nat}(\mathsf{rem}(\mathit{dvd}, \mathit{dvs})) < \mathsf{bv2nat}(\mathit{dvs}). \tag{7}$$

The restriction to non-null divisors is relevant, because:

$$\mathsf{div}(\mathit{dvd}, \overrightarrow{0}) = \overrightarrow{1} \quad \text{and} \quad \mathsf{rem}(\mathit{dvd}, \overrightarrow{0}) = \mathit{dvd}.$$

Division and remainder in Java throw an exception when the divisor is null. This behaviour is realised in the semantics used by the LOOP tool via a wrapper function around the bitvector operations that we are describing. But this wrapper is omitted here.

These two results (6) and (7) characterise division and remainder, in the following sense.

$$\begin{aligned}&\forall q, r \in \mathbb{N}.\ \mathsf{bv2nat}(\mathit{dvs}) * q + r = \mathsf{bv2nat}(\mathit{dvd}) \wedge r < \mathsf{bv2nat}(\mathit{dvs})\\ &\quad \Longrightarrow q = \mathsf{bv2nat}(\mathsf{div}(\mathit{dvd}, \mathit{dvs})) \wedge r = \mathsf{bv2nat}(\mathsf{rem}(\mathit{dvd}, \mathit{dvs})).\end{aligned} \tag{8}$$

This is, together with (6) and (7), the main result of this section. It allows us to prove various results about (unsigned) division and remainder, such as:

$$\mathsf{div}(\mathit{dvd}, \mathbf{1}) = \mathit{dvd} \quad \text{and} \quad \mathsf{rem}(\mathit{dvd}, \mathbf{1}) = \overrightarrow{0}.$$

And:

$$\mathsf{bv2nat}(\mathsf{div}(\mathit{dvd}, \mathit{dvs})) = 0 \Longleftrightarrow \mathsf{bv2nat}(\mathit{dvd}) < \mathsf{bv2nat}(\mathit{dvs}).$$

7 Signed division and remainder

Our aim in this section is first to introduce signed division and remainder operations, and prove the analogues of (6), (7) and (8). Next we intend to prove the properties that are listed in the Java Language Specification [8] about division and remainder.

Before we move from unsigned division and remainder to the signed versions (as used in Java), we recall that:

$$\begin{array}{rcr} 5 \;/\; 3 = 1 & \qquad & 5 \;\%\; 3 = 2 \\ 5 \;/\; -3 = -1 & & 5 \;\%\; -3 = 2 \\ -5 \;/\; 3 = -1 & & -5 \;\%\; 3 = -2 \\ -5 \;/\; -3 = 1 & & -5 \;\%\; -3 = -2. \end{array}$$

In line with these results, we make the following case distinctions:

$$\begin{array}{ll}
dvd \;/\; dvs & dvd \;\%\; dvs \\
\;\stackrel{\text{def}}{=}\; \text{if } \mathsf{bv2int}(dvd) \geq 0 & \;\stackrel{\text{def}}{=}\; \text{if } \mathsf{bv2int}(dvd) \geq 0 \\
\quad \text{then if } \mathsf{bv2int}(dvs) \geq 0 & \quad \text{then if } \mathsf{bv2int}(dvs) \geq 0 \\
\qquad \text{then } \mathsf{div}(dvd, dvs) & \qquad \text{then } \mathsf{rem}(dvd, dvs) \\
\qquad \text{else } -\mathsf{div}(dvd, -dvs)) & \qquad \text{else } \mathsf{rem}(dvd, -dvs)) \\
\quad \text{else if } \mathsf{bv2int}(dvs) \geq 0 & \quad \text{else if } \mathsf{bv2int}(dvs) \geq 0 \\
\qquad \text{then } -\mathsf{div}(-dvd, dvs)) & \qquad \text{then } -\mathsf{rem}(-dvd, dvs)) \\
\qquad \text{else } \mathsf{div}(-dvd, -dvs) & \qquad \text{else } -\mathsf{rem}(-dvd, -dvs)
\end{array}$$

Using the properties of unsigned division and remainder we quickly get:

$$dvd \;/\; \overrightarrow{0} = \begin{cases} \overrightarrow{1} & \text{if } \mathsf{bv2int}(dvd) \geq 0 \\ \mathbf{1} & \text{otherwise.} \end{cases} \qquad dvd \;\%\; \overrightarrow{0} = dvd$$

$$dvd \;/\; \mathbf{1} = dvd \qquad dvd \;\%\; \mathbf{1} = \overrightarrow{0}.$$

The signed analogue of (7) involves the absolute value function:

$$\mathsf{bv2int}(dvs) \neq 0 \Longrightarrow \mathsf{abs}(\mathsf{bv2int}(dvd \;\%\; dvs)) < \mathsf{abs}(\mathsf{bv2int}(dvs)). \tag{9}$$

The analogue of (6) involves an overflow exception:

$$\begin{array}{l} \neg\big(\mathsf{bv2int}(dvd) = -2^{N-1} \wedge \mathsf{bv2int}(dvs) = -1\big) \\ \quad \Longrightarrow \\ \mathsf{bv2int}(dvd) = \mathsf{bv2int}(dvs) * \mathsf{bv2int}(dvd \;/\; dvs) + \mathsf{bv2int}(dvd \;\%\; dvs). \end{array} \tag{10}$$

The proof of this property is obtained from (6), applied after the various case distinctions. The overflow case—when $dvd = \mathsf{minint}$ and $dvs = \overrightarrow{1}$—does not satisfy (10) because:

$$\mathsf{bv2int}(\mathsf{minint} \;/\; \overrightarrow{1}) = -2^{N-1} \qquad \mathsf{bv2int}(\mathsf{minint} \;\%\; \overrightarrow{1}) = 0.$$

Actually, we can move the bv2int's in (10) to the outside and remove them (because bv2int is injective). This yields:

$$dvd = dvs * \big(dvd \,/\, dvs\big) + \big(dvd \,\%\, dvs\big). \tag{11}$$

The restriction from (10) disappears in this form—where $*$ is multiplication from Section 5, with its own overflow behaviour.

Next we turn to the sign of signed division and remainder. It is most complicated for division.

$$\begin{array}{l}
\neg\big(\mathsf{bv2int}(dvd) = -2^{N-1} \wedge \mathsf{bv2int}(dvs) = -1\big) \;\wedge\; \mathsf{bv2int}(dvs) \neq 0 \\
\quad\Longrightarrow \\
\Big(\mathsf{bv2int}(dvd \,/\, dvs) > 0 \Leftrightarrow (\mathsf{bv2int}(dvd) \geq \mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) > 0) \\
\qquad\qquad\qquad\qquad\quad \vee\; (\mathsf{bv2int}(dvd) \leq \mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) < 0)\Big) \\
\wedge \\
\Big(\mathsf{bv2int}(dvd \,/\, dvs) = 0 \Leftrightarrow \mathsf{abs}(\mathsf{bv2int}(dvd)) < \mathsf{abs}(\mathsf{bv2int}(dvs))\Big) \\
\wedge \\
\Big(\mathsf{bv2int}(dvd \,/\, dvs) < 0 \Leftrightarrow (\mathsf{bv2int}(dvd) \geq -\mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) < 0) \\
\qquad\qquad\qquad\qquad\quad \vee\; (\mathsf{bv2int}(dvd) \leq -\mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) > 0)\Big)
\end{array} \tag{12}$$

About the sign of the remainder we can only say it is determined by the sign of the dividend:

$$\begin{array}{l}
\mathsf{bv2int}(dvd) > 0 \Longrightarrow \mathsf{bv2int}(dvd \,\%\, dvs) \geq 0 \\
\mathsf{bv2int}(dvd) < 0 \Longrightarrow \mathsf{bv2int}(dvd \,\%\, dvs) \leq 0.
\end{array} \tag{13}$$

The uniqueness of signed division and remainder requires more assumptions than in the unsigned case (8). It involves for instance the above sign descriptions.

$$\begin{array}{l}
\forall q, r \in \mathbb{Z}.\; \neg\big(\mathsf{bv2int}(dvd) = -2^{N-1} \wedge \mathsf{bv2int}(dvs) = -1\big) \;\wedge \\
\qquad \mathsf{bv2int}(dvs) \neq 0 \;\wedge \\
\qquad \Big(q > 0 \Leftrightarrow (\mathsf{bv2int}(dvd) \geq \mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) > 0) \\
\qquad\qquad\quad \vee\; (\mathsf{bv2int}(dvd) \leq \mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) < 0)\Big) \wedge \\
\qquad \Big(q = 0 \Leftrightarrow \mathsf{abs}(\mathsf{bv2int}(dvd)) < \mathsf{abs}(\mathsf{bv2int}(dvs))\Big) \wedge \\
\qquad \Big(q < 0 \Leftrightarrow (\mathsf{bv2int}(dvd) \geq -\mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) < 0) \\
\qquad\qquad\quad \vee\; (\mathsf{bv2int}(dvd) \leq -\mathsf{bv2int}(dvs) \wedge \mathsf{bv2int}(dvs) > 0)\Big) \wedge \\
\qquad \Big(\mathsf{bv2int}(dvd) > 0 \Rightarrow r \geq 0\Big) \wedge \\
\qquad \Big(\mathsf{bv2int}(dvd) < 0 \Rightarrow r \leq 0\Big) \wedge \\
\qquad \mathsf{abs}(r) < \mathsf{abs}(\mathsf{bv2int}(dvs)) \wedge \\
\qquad \mathsf{bv2int}(dvs) * q + r = \mathsf{bv2int}(dvd) \\
\qquad\quad \Longrightarrow \\
\qquad q = \mathsf{bv2int}(dvd \,/\, dvs) \wedge r = \mathsf{bv2int}(dvd \,\%\, dvs).
\end{array} \tag{14}$$

As consequence, we obtain the relation between widening and division & remainder, following (3) for addition and (4) for multiplication.

$$\begin{aligned}&\neg\big(\mathsf{bv2int}(\mathit{dvd}) = -2^{N-1} \wedge \mathsf{bv2int}(\mathit{dvs}) = -1\big)\\ &\quad\Longrightarrow \mathsf{widen}(\mathit{dvd}) \;/\; \mathsf{widen}(\mathit{dvs}) = \mathsf{widen}(\mathit{dvd} \;/\; \mathit{dvs}) \wedge\\ &\quad\ \mathsf{widen}(\mathit{dvd}) \,\%\, \mathsf{widen}(\mathit{dvs}) = \mathsf{widen}(\mathit{dvd} \,\%\, \mathit{dvs}).\end{aligned} \tag{15}$$

As another consequence we can relate division and remainder for bitvectors to their mathematical counterparts (for integers). In order to do so we use the floor and fractional functions[5] from the standard PVS library. For a real x, $\mathsf{floor}(x)$ is the unique integer i with $i \leq x < i + 1$. And $\mathsf{fractional}(x)$ is then $x - \mathsf{floor}(x)$, which is in the interval $[0, 1)$. The main result is split in two parts. The difference is in the "$+1$" and "-1" in the last two lines.

$$\begin{aligned}&\neg\big(\mathsf{bv2int}(\mathit{dvd}) = -2^{N-1} \wedge \mathsf{bv2int}(\mathit{dvs}) = -1\big) \;\wedge\\ &\quad\big(\,(\mathsf{bv2int}(\mathit{dvd}) = 0 \wedge \mathsf{bv2int}(\mathit{dvs}) \neq 0) \;\vee\\ &\quad\ \ (\mathsf{bv2int}(\mathit{dvd}) > 0 \wedge \mathsf{bv2int}(\mathit{dvs}) > 0) \;\vee\\ &\quad\ \ (\mathsf{bv2int}(\mathit{dvd}) < 0 \wedge \mathsf{bv2int}(\mathit{dvs}) < 0) \;\vee\\ &\quad\ \ (\mathsf{bv2int}(\mathit{dvs}) \neq 0 \wedge \exists n \in \mathbb{Z}.\, \mathsf{bv2int}(\mathit{dvd}) = n * \mathsf{bv2int}(\mathit{dvs}))\,\big)\\ &\Longrightarrow \mathsf{bv2int}(\mathit{dvd} \;/\; \mathit{dvs}) = \mathsf{floor}(\mathsf{bv2int}(\mathit{dvd}) \,/\, \mathsf{bv2int}(\mathit{dvs})) \;\wedge\\ &\quad\ \ \mathsf{bv2int}(\mathit{dvd} \,\%\, \mathit{dvs}) = \mathsf{bv2int}(\mathit{dvs}) * \mathsf{fractional}(\mathsf{bv2int}(\mathit{dvd}) \,/\, \mathsf{bv2int}(\mathit{dvs})).\end{aligned}$$

$$\begin{aligned}&\big(\,(\mathsf{bv2int}(\mathit{dvd}) > 0 \wedge \mathsf{bv2int}(\mathit{dvs}) < 0) \;\vee\\ &\ \ (\mathsf{bv2int}(\mathit{dvd}) < 0 \wedge \mathsf{bv2int}(\mathit{dvs}) > 0)\,\big) \;\wedge\\ &\neg\exists n \in \mathbb{Z}.\, \mathsf{bv2int}(\mathit{dvd}) = n * \mathsf{bv2int}(\mathit{dvs}))\\ &\Longrightarrow \mathsf{bv2int}(\mathit{dvd} \;/\; \mathit{dvs}) = \mathsf{floor}(\mathsf{bv2int}(\mathit{dvd}) \,/\, \mathsf{bv2int}(\mathit{dvs})) + 1 \;\wedge\\ &\quad\ \ \mathsf{bv2int}(\mathit{dvd} \,\%\, \mathit{dvs}) = \mathsf{bv2int}(\mathit{dvs}) * \big(\mathsf{fractional}(\mathsf{bv2int}(\mathit{dvd}) \,/\, \mathsf{bv2int}(\mathit{dvs})) - 1\big).\end{aligned}$$

Such results are used as definitions in [16].

7.1 Division in Java

We start with a quote from the Java Language Specification [8, §§15.17.2].

> Integer division rounds toward 0. That is, the quotient produced for operands n and d that are integers after binary numeric promotion (§5.6.2) is an integer value q whose magnitude is as large as possible while satisfying $|d * q| \leq |n|$; moreover, q is positive when and n and d have the same sign, but q is negative when and n and d have opposite signs. There is one special case that does not satisfy this rule: if the dividend is the negative integer of largest possible magnitude for its type, and the divisor is -1, then integer overflow occurs and the result is equal to the dividend. Despite the overflow, no exception is thrown in this case. On the other hand, if the value of the divisor in an integer division is 0, then an ArithmeticException is thrown.

[5] Developed by Paul Miner.

We check that all these properties hold for our signed division and remainder operations defined on bitvectors in PVS. The first property stating that the quotient is "…as large as possible …" is formalised (and proven) as:

$$\begin{array}{l} \forall q\colon \mathbb{Z}.\ \ \mathsf{bv2int}(dvs) \neq 0 \wedge \mathsf{abs}(\mathsf{bv2int}(dvs) * q) \leq \mathsf{abs}(\mathsf{bv2int}(dvd)) \\ \qquad\qquad \Longrightarrow \mathsf{abs}(q) \leq \mathsf{abs}(\mathsf{bv2int}(dvd\ /\ dvs)). \end{array}$$

The sign of the quotient has already been described in (12). And the "…one special case …" in this quote refers to the assumption in (10).

7.2 Remainder in Java

The relevant quote [8, §§15.17.3] says:

> The remainder operation for operands that are integers after binary numeric promotion (§5.6.2) produces a result value such that $(a/b) * b + (a\%b)$ is equal to a. This identity holds even in the special case that the dividend is the negative integer of largest possible magnitude for its type and the divisor is -1 (the remainder is 0). It follows from this rule that the result of the remainder operation can be negative only if the dividend is negative, and can be positive only if the dividend is positive; moreover, the magnitude of the result is always less than the magnitude of the divisor. If the value of the divisor for an integer remainder operator is 0, then an ArithmeticException is thrown.

The identity "$(a/b)*b+(a\%b)$ is equal to a" in this quote holds as (11), indeed without the restriction that occurs in (10). The statement about the sign of the remainder is stated in (13), and about its magnitude in (9).

We conclude that all properties of division and remainder required in the Java Language Specification hold for our formalisation in PVS.

8 Conclusions

This paper formalises the details of multiplication, division and remainder operations for bitvectors in the higher order logic of the theorem prover PVS, and makes precise which properties that this formalisation satisfies. This is typical theorem prover work, involving many subtle details and case distinctions (which humans easily get wrong). The main application area is Java program verification. Therefore, the relation between the newly defined bitvector operations and Java's widening and narrowing functions gets much attention.

The theories underlying this paper have recently been included (by Sam Owre) in the bitvector library of PVS version 3.0 (and upwards). Also, the bitvector semantics is now heavily used for verifying specific Java programs, see for instance [11].

Acknowledgements

Thanks are due to Joseph Kiniry and Erik Poll for their feedback on this paper.

References

1. W. Ahrendt, Th. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P.H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Knutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering (FASE)*, number 2306 in Lect. Notes Comp. Sci., pages 327–330. Springer, Berlin, 2002.
2. B. Beckert and S. Schlager. Integer arithmetic in the specification and verification of Java programs. Workshop on Tools for System Design and Verification (FM-TOOLS), `http://i12www.ira.uka.de/~beckert/misc.html`, 2002.
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. V.A. Carreño and P.S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In E.Th. Schubert, Ph.J. Windley, and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications*, 1995. Category B Proceedings.
5. P. Chalin. Improving jml: For a safer and more effective language. In *Formal Methods Europe 2003*, 2003, to appear.
6. Z. Chen. *Java Card Technology for Smart Cards*. The Java Series. Addison-Wesley, 2000.
7. 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.
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, 2000.
9. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Theorem Proving in Higher Order Logics*, number 1690 in Lect. Notes Comp. Sci., pages 113–130. Springer, Berlin, 1999.
10. J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, number 1869 in Lect. Notes Comp. Sci., pages 234–251. Springer, Berlin, 2000.
11. B. Jacobs, M. Oostdijk, and M. Warnier. Formal verification of a secure payment applet. *Journ. of Logic and Algebraic Programming*, To appear.
12. 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.
13. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual (draft). `www.jmlspecs.org`, 2003.
14. C. Marché, C. Paulin, and X. Urbain. The Krakatoa tool for certification java/javacard programs annotated in jml. *Journ. of Logic and Algebraic Programming*, To appear.
15. S. Owre, J.M. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Trans. on Softw. Eng.*, 21(2):107–125, 1995.
16. N. Rauch and B. Wolff. Formalizing Java's two's-complement integral type in Isabelle/HOL. In Th. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS'03)*, number 80 in Elect. Notes in Theor. Comp. Sci. Elsevier, Amsterdam, 2003. `www.elsevier.nl/locate/entcs/volume80.html`.
17. W. Stallings. *Computer Organization and Architecture*. Prentice Hall, 4^{th} edition, 1996.
18. L. Théry. A library for floating-point numbers in Coq. `www-sop.inria.fr/lemme/AOC/coq/`, 2002.
19. J.F.H. Winkler. A safe variant of the unsafe integer arithmetic of Java™. *Software—Practice and Experience*, 33:669–701, 2002.