Analysis and formal specification of OpenJDK’s BitSet

Tatman, A.S.; Hiep, H.A.; Gouw, C.P.T. de; Herber, P.; Wijs, A.

Analysis and Formal Specification of OpenJDK's BitSet

Andy S. Tatman[1(✉)], Hans-Dieter A. Hiep[1,2], and Stijn de Gouw[3]

[1] Leiden Institute of Advanced Computer Science (LIACS), Leiden, The Netherlands
tatmanandys@gmail.com, hdh@cwi.nl
[2] Centrum Wiskunde and Informatica, Amsterdam, The Netherlands
[3] Open Universiteit, Heerlen, The Netherlands
sdg@ou.nl

Abstract. This paper uses a combination of formal specification and testing, to analyse OpenJDK's `BitSet` class. This class represents a vector of bits that grows as required. During our analysis, we uncovered a number of bugs. We propose and compare various solutions, supported by our formal specification. While a full mechanical verification of the `BitSet` class is not yet possible due to limited support for bitwise operations in the KeY theorem prover, we show initial steps taken to formally verify the challenging `get(int,int)` method, and discuss some required extensions to the theorem prover.

Keywords: Formal specification · Testing · Java/OpenJDK · KeY · JML

1 Introduction

Formal specification and verification are extremely powerful techniques to inspect program code and determine either its correctness or find errors that can be missed by traditional testing techniques. These formal methods may uncover bugs that have laid dormant in code for years. However, applying formal methods can also be extremely time-consuming: even a small section of code can require a large proof to verify it. As such, formal verification is generally directed to essential and frequently used code, such as standard libraries. Previous examples of such an effort include the verification of OpenJDK's `LinkedList` class [12] and OpenJDK's sorting implementation [10]. In this paper, we discuss and analyse another of Java's standard library classes, specifically the OpenJDK's `BitSet` class. The original goal was to formally verify the correctness of an essential part of the `BitSet` class using the KeY theorem prover. However, when using techniques such as formal specification and testing, we encountered a number of issues that appear to have existed in the code since the original push on OpenJDK's public repository back in 2007[1].

[1] https://github.com/openjdk/jdk/blob/319a3b994703aac84df7bcde272adfcb3cdbbbf0/jdk/src/share/classes/java/util/BitSet.java.

P. Herber and A. Wijs (Eds.): iFM 2023, LNCS 14300, pp. 134–152, 2024.
https://doi.org/10.1007/978-3-031-47705-8_8

We first identified an overflow bug in `BitSet`'s `get(int,int)` method. Later, we also encountered issues with the `valueOf(...)` methods that under certain conditions leaves (an instance of) bitset in an unexpected state, causing erratic behaviour in the other methods of the class. We have chosen to use the KeY theorem prover because it most accurately models Java semantics, including, for example, integer overflows. Unlike other available verification tools, KeY allows to load the unaltered `BitSet` class. And even then, we need to extend KeY with additional proof rules before we are able to perform a full verification. However, a full verification of the `BitSet` class is not yet possible, since the issues we encountered are not yet resolved by the Java developers and so the specification and implementation is not settled yet.

Related Work. To the best of our knowledge, this is the first paper presenting a formal analysis of Java's `BitSet` class, but there is related work in two directions. On the one hand, in recent years there have been several case studies in formal verification [2,6,13] and model checking [3,9] of various Java libraries. However, these libraries did not substantially use bitwise operations. At most a few bit-shifts were present and shifts can be covered purely arithmetically in a fairly straightforward manner by multiplying or dividing with a power of two.

In another direction, there are numerous works that focus on (mechanisation of) logical theories for bit vectors, not necessarily tied to Java. The SMT solver Z3 [15] has a theory for fixed-width bit vectors. It works roughly by flattening (also known as bit-blasting) a given arithmetic formula of interest that involves bit vectors into an equisatisfiable propositional formula and then solving the resulting propositional formula with SAT-solving techniques. An extension of the CVC4 SMT-solver [11] also supports bit vectors using bit-blasting and, recently, a more advanced technique called int-blasting [17]. Isabelle/HOL is a proof assistant that supports bit vectors [7], building on the work in Z3. The Coq proof assistant also includes a theory for bit vectors [5] which has been applied to a (self-written) library for finite sets, represented by bit vectors. There is also a tool-supported approach for verifying LTL properties (a common temporal logic) of programs involving bit vectors.

While none of these solvers and proof assistants directly support the full-fledged Java semantics required to load and analyse the unaltered `BitSet` class with the formal JML specifications, they could potentially serve as back-ends to solve proof obligations that arise during verification with KeY. This requires developing a translator for proof obligations from KeY into e.g. SMT-LIB. KeY already supports translating standard arithmetic formulas into SMT-LIB (and then using e.g. Z3 as a back-end) but the translation of bitwise operations is limited and would have to be enhanced: most bitwise operations are currently translated as uninterpreted function symbols.

Outline. In Sect. 2 we explain the structure and inner workings of the `BitSet` class. Section 3 then discusses our formal specification that captures the expected behaviour of the class. Section 4 discusses the issues that we discovered while analysing the correctness of the class, and Sect. 4.3 offers various solution directions. Finally, Sect. 5 covers a proof sketch of the formal verification of the `get(int,int)` method, as well as extensions that are required to the KeY theorem prover in order to complete the proof.

Listing 1. The fields and methods of the `BitSet` class relevant for this paper. See also the Javadoc of `BitSet` [1] for a full description of all its methods.

```
package java.util;

public class BitSet {
  // The internal field storing the bits.
  private long[] words;
  // The number of words in the logical size of this BitSet.
  private transient int wordsInUse = 0;

  /** Creates a new bit set. */
  public BitSet() { ... }
  /** Creates a bit set whose initial size is large enough to
     explicitly represent bits with indices in the range 0 through
     nbits-1. */
  public BitSet(int nbits) { ... }
  /** Returns a new bit set containing all the bits in the given long
     array. */
  public static BitSet valueOf(long[] longs) { ... }

  /** Sets the bit at the specified index to true. */
  public void set(int bitIndex) { ... }
  /** Returns the value of the bit with the specified index. */
  public boolean get(int bitIndex) { ... }
  /** Sets the bit specified by the index to false. */
  public void clear(int bitIndex) { ... }
  /** Returns a new BitSet composed of bits from this BitSet from
     fromIndex (inclusive) to toIndex (exclusive). */
  public BitSet get(int fromIndex, int toIndex) { ... }

  /** Returns the "logical size" of this BitSet: the index of the
     highest set bit in the BitSet plus one. */
  public int length() { ... }
}
```

2 The `BitSet` Class

The `BitSet` class is part of Java's standard library in the open-source Java Development Kit (OpenJDK). Listing 1 shows the fields and methods of the class relevant for this paper. The class allows users to store bits (or primitive Booleans) as a bit vector and packs these bits efficiently as an array of elements of primitive type `long`, where each long element stores (and occupies, on mainstream architectures) 64 bits. This is typically far more efficient memory-wise than storing an unpacked array of individual primitive Booleans. In arrays, all elements must be directly addressable, and so, on byte-aligned memory architectures, every single bit in an array of primitive Booleans would use 8 bits.

The class has methods to set, clear or get the value of one bit, as well as methods to do the same for sequences of consecutive bits. These methods operate on Booleans, and internally perform packing and unpacking of the bit vector. We shall simply speak of bit values 1 and 0, instead of true and false, respectively.

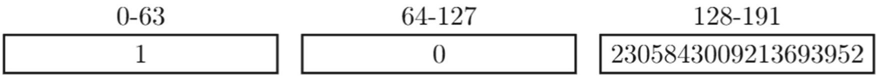

Fig. 1. A representation of the `words` array. Each individual word is depicted by a decimal number inside a box. The third box contains the decimal number 2^{61}, which has exactly 1 bit set to 1. `wordsInUse` is 3, as the `words` array has 3 elements and the last word has bits set.

Fig. 2. The logical representation of the same bitset as depicted in Fig. 1. Each bit is stored separately. Every bit between the dots is set to 0. The bit in 189 is set to 1, because it is the bit set in 2^{61} in the third element of `words`.

The field `words` contains the array of (64-bit) long elements. Each word packs bits, also making use of the sign bit. Index 0 of a bitset is the least significant bit in the first word, index 63 is the most significant bit of the first word (the sign bit), and index 64 is the least significant bit of the second word.

Figure 1 shows the `words` array of a bitset instance, while Fig. 2 shows the logical representation of that same array as a sequence of bits. The class also maintains an integer field `wordsInUse` that keeps track of the last word that contains at least one set bit. The `wordsInUse` field is used to approximate the logical size of the bitset instance. In fact, the *logical size* of the `BitSet` is the position of the most significant bit that is set to 1, and therefore is closely related to `wordsInUse`. If no bits are set in a bitset instance, then the logical size is 0. If the first bit (at index 0) is the most significant bit that is set to 1, the logical size is 1. In the example above, the logical size is 190, as index 189 is the last bit that is set to 1.

Initially every bit in a bitset instance is set to 0. If a user tries to retrieve the value of a bit outside of the logical size of a bitset, then this value is by default 0. This allows the class to handle access to any bit at a non-negative index, even if the corresponding index would fall outside the bounds of the `words` array. When setting a bit at an index outside of the `words` array, the bitset expands dynamically by allocating a larger `words` array.

3 Formal Specification

We focused on a selection of methods that cover the main operations of the `BitSet` class: querying and modifying bitsets, shown in Listing 1, and the internal methods `recalculateWordsInUse()`, `expandTo(int)`, `ensureCapacity(int)` that are explained later. Next, we formulate a specification of the class, in the form of a class invariant and contracts for methods in scope. We also introduce model methods (see below) to express method contracts at an abstraction level that corresponds more closely to our intuition of expected method behaviour.

We employ the Java Modelling Language (JML) [14] as the language in which we express our formal specification. The KeY tool automatically translates our given specifications into Java Dynamic Logic (JavaDL) [2] to be able to reason about the correctness of methods.

Method contracts describe what must be true in the state prior to the method being called (pre-condition) and what must be true in the state after the method terminates (post-condition). The pre-condition for a method is described in JML using the `requires` clause, while the post-condition is described using the `ensures` clause. A contract can also specify exactly what parts of the heap can be altered using the `assignable` clause. For example, `assigable \nothing` means that the fields of any pre-existing object must remain exactly the same, but the method is allowed to create new objects.

Further, we distinguish helper methods from normal methods. The contracts of normal methods implicitly includes the class invariant as part of the pre-condition and post-condition, while helper methods do not implicitly include the class invariant. We can use `\invariant_for(this)` to explicitly specify that the invariant *does* hold in the pre-condition or post-condition of a helper method. This is used, for example, in the helper method `recalculateWordsInUse()` that restores the class invariant of a bitset where the invariant did not hold in the pre-condition.

3.1 Class Invariant

Our starting point for defining the class invariant is the three assertions given in the `checkInvariants()` method. These are the following:

1. Either `wordsInUse` is zero or `words[wordsInUse-1]` is non-zero. The latter condition states that the word possibly indicated by `wordsInUse` has at least one bit that is set.
2. The value of `wordsInUse` is in the range of [0, `words.length`], inclusive.
3. Either `wordsInUse` equals the length of `words`, or the first word outside the meaningful part of the `words` array, i.e. `words[wordsInUse]`, has no set bits and so `words[wordsInUse] = 0`.

These conditions are indeed necessarily part of the class invariant, but these conditions alone are not sufficient: there are more conditions that remain invariant.

The `words` array is allocated and we know that `words` is never null. Further, the last condition suggests that *all* words after `words[wordsInUse-1]` should be equal to zero. In fact, the implementations of the in-scope methods guarantee this property. As an example, take the `recalculateWordsInUse` method. This helper method restores the class invariant by setting the `wordsInUse` variable to the proper value: when the method is called, it is assumed that all words after `words[wordsInUse-1]` equal zero, and the method moves the `wordsInUse` as much as is possible to the left to ensure that condition (1) above holds. By moving `wordsInUse` to the left when `words[wordsInUse]` is zero, we indeed have that all words after `wordsInUse` are equal to 0.

We formalise this intuition by the class invariant in Listing 2.

Listing 2. The first part of the class invariant, written in JML.

```
/*@ invariant
  @ words != null &
  @ // The first three are from checkInvariants:
  @ (wordsInUse == 0 | words[wordsInUse - 1] != 0) &&
  @ (wordsInUse >= 0 && wordsInUse <= words.length) &&
  @ (wordsInUse == words.length || words[wordsInUse] == 0) &&
  @ // Our addition to the invariant:
  @ (wordsInUse < words.length ==> (\forall \bigint i; wordsInUse <= i
      < words.length; words[i] == 0) ) &&
  @ ...
  @*/
```

Next, we look for upper bounds of `words.length` and `wordsInUse`. Bitsets that are generated by the public constructors (i.e. not by the `valueOf(...)` methods, see Sect. 4.2) will allocate a `words` array. When acting on bitsets using e.g. the `set(...)` method, the `words` array grows as required by the internal `expandTo(int)` and `ensureCapacity(int)` methods, while the `wordsInUse` variable is updated to reflect the largest word with a set bit. The largest addressable position of a bit is at position `Integer.MAX_VALUE`, which is stored in `words[Integer.MAX_VALUE/64]`. Hence, the upper bound of `wordsInUse` is `Integer.MAX_VALUE/64 + 1`.

The `ensureCapacity(int wordsRequired)` method grows the array if necessary, specifically if `wordsRequired` is larger than the current length of `words`. If the array should grow, this method allocates a new array of length `Math.max(2 * words.length, wordsRequired)`, meaning that the array gets at least doubled every time `words` is expanded. The bound for the parameter `wordsRequired` is the same as for `wordsInUse`, namely `Integer.MAX_VALUE/64 + 1`. The largest `word` array that the public constructors create is also of length `Integer.MAX_VALUE/64 + 1`. For the upper bound of the length of `words`, we thus take double this value: `2 * (Integer.MAX_VALUE/64 + 1)`.

These bounds hold while using `BitSet`'s methods to interact with specific bits, such as `set(...)` and `clear(...)`. However, in Sect. 4.2, we will show that these bounds are violated when using the static `valueOf(...)` methods.

3.2 The `wordsToSeq()` Model Method

In order to express properties of the contents of a bitset, we use a sequence of Booleans as representation, such that position i in the sequence corresponds to the bit at position i in the bitset. We employ a *model method*, which is a method that is only used in our contracts and does not affect the (run-time) state of the object [8], shown in Listing 3.

Listing 3. Our wordsToSeq() model method.

```
/*@ private model strictly_pure \seq wordsToSeq() {
  @  return (\seq_def \bigint i; 0; (\bigint)wordsInUse * (\bigint)
     BITS_PER_WORD; (words[i / BITS_PER_WORD] >>>
              (int)(i % BITS_PER_WORD)) & 1);
  @ }
  @*/
```

For each word in the `words` array, the sequence isolates each of the individual 64 bits and stores them as an element of the sequence returned by the model method. Note that, contrary to the logical size of a bitset, the length of our sequence of Booleans is a multiple of 64, the number of bits per word. As an example, consider that the `wordsToSeq()` model method converts the array as seen in Fig. 1 to the sequence as seen in Fig. 2.

As with the behaviour of the `BitSet` class itself, any bit at a position larger than the length of this sequence in the `words` array must equal 0.

It is now possible to give contracts for the methods `get(int)`, `set(int)`, and `clear(int)`. Namely, the value that is returned by `get(int)` is precisely the value of the Boolean of the `wordsToSeq` sequence at the right position, or zero if it falls outside. Similarly, for `set(int)` and `clear(int)` we can relate the `wordsToSeq` sequence in the pre-state and the post-state by expressing what bit values remain unchanged, and the new bit value at the changed position in the bit vector.

3.3 The `get(int,int)` Method

A more challenging method to specify is the `get(int fromIndex, int toIndex)` method. It returns a new `BitSet` instance that contains the bits from the given range. As we will show in Sect. 4, the `get(int,int)` method has a bug in it, not only in the current Java version (JDK 20, at the time of writing)[2] but also all the way back to the first release of OpenJDK and possibly even further back. Assuming that the bug will eventually be resolved, `get(int,int)` is still an interesting method to look at. It is one of the larger and more complex methods in the `BitSet` class, and its verification requires giving a non-trivial loop invariant.

The method returns a subsequence of the current bitset, containing all bits from the `fromIndex` up to but *not* including the `toIndex`. Both `fromIndex` and `toIndex` must be non-negative integers, and `fromIndex` must be less than or equal `toIndex`. Furthermore, the specification involves comparing two different Boolean sequences, namely the original sequence and the sequence associated to the new `BitSet` instance returned by the method.

The contract for this method can be seen in Listing 4.

Listing 4. The contract for the `get(int,int)` method.

```
/*@ normal_behaviour
  @ requires fromIndex >= 0 && fromIndex <= toIndex;
  @ ensures \result != this && \invariant_for(\result);
  @ ensures (\forall \bigint i; 0 <= i < \result.wordsToSeq().length;
     (fromIndex + i < wordsToSeq().length ?
     wordsToSeq()[fromIndex + i] : 0) == \result.wordsToSeq()[i]);
  @ ensures (\result.wordsToSeq().length < toIndex - fromIndex) ==>
     (\forall \bigint i; \result.wordsToSeq().length <= i < toIndex -
     fromIndex; (fromIndex + i < wordsToSeq().length ?
      wordsToSeq()[fromIndex + i] : 0) == 0);
```

[2] https://github.com/openjdk/jdk/blob/a52c4ede2f043b7d4a234c7d06f91871312e9654/src/java.base/share/classes/java/util/BitSet.java.

```
  @ assignable \nothing;
  @*/
```

The pre-condition of the method states that $0 \leq$ `fromIndex` $\leq$ `toIndex`. For the post-condition of this method, we have, first of all, that the invariant must hold for the resulting bitset and that the resulting instance is different from `this`. Further, the last two `ensures` clauses express that the resulting bitset contains the expected bits. Every element in `result.wordsToSeq()` should match in value to the corresponding element in the original `this.wordsToSeq()`. If an element at position i is out of the bounds of one of the Boolean sequences, then that element should equal 0 in the other sequence. For example, assume the user calls `get(0, 100)` and the method returns a bitset with `result.wordsToSeq().length` $= 64$. This means that the bits at positions 64–99 in `result` are set to 0, and as such the corresponding bits in the original bitset should also all equal 0. Finally, the `assignable \nothing` clause expresses that the state of the current object is not changed in any way.

4 Issues in **BitSet**

Using formal specification and testing, we discovered several issues. These issues are outlined in this section, and we suggest solution directions. The two issues are orthogonal, but the issues do overlap in one aspect: an integer overflow of the logical size as returned by `length()`.

4.1 A Bug in **get(int,int)** Caused by a Negative **length()**

The first issue occurs in the `get(int fromIndex, int toIndex)` method[3]. The beginning of the implementation of this method is visible in Listing 5.

Listing 5. Beginning of the `get(int, int)` method, where the first bug occurs.

```java
public BitSet get(int fromIndex, int toIndex) {
  checkRange(fromIndex, toIndex);
  checkInvariants();
  int len = length();
  if (len <= fromIndex || fromIndex == toIndex)
    return new BitSet(0); // If no set bits in range
  if (toIndex > len)
    toIndex = len; // An optimization
  ...
```

The `length()` method should return the position of the most significant bit set, plus 1. For example, if the user sets the bit at position 200 in a previously empty bitset, then the `length()` method will return 201. However, if the user sets the bit at index `Integer.MAX_VALUE`, then the `length()` method will return the integer `Integer.MAX_VALUE` $+ 1$, which overflows to `Integer.MIN_VALUE`.

Listing 6. Example of how the bug can lead to unexpected results of `get(int,int)`.

```java
BitSet bset = new BitSet(0);
```

[3] This bug report has been accepted by Oracle, see JDK-8305734.

```
bset.set(Integer.MAX_VALUE);
bset.set(999);
BitSet result = bset.get(0,1000);
```

Listing 6 shows an example where this gives faulty behaviour. The expected behaviour would be that `result` is a bitset with logical size 1000 and which has bit 999 set. However, with the current implementation, the `result` has logical size 0 and has no bits set!

This is because `bset.length()` returns the negative `Integer.MIN_VALUE`. The expression `len <= fromIndex` on line 5 will always evaluate to true, since `Integer.MIN_VALUE` is smaller than or equal to all 32-bit signed integers, causing the `bset.get(0,1000)` to return the empty bitset.

4.2 Bugs Caused by `valueOf(...)` Corrupting `length()`

The next issue occurs in the `valueOf(...)` methods[4]. We focus on the method with a parameter of type `long[]` (Listing 7), but the same bug occurs in the overloaded methods with parameter types `LongBuffer`, `ByteBuffer` and `byte[]`.

Listing 7. The `valueOf(long[])` method and the private constructor it uses.

```
private BitSet(long[] words) {
  this.words = words;
  this.wordsInUse = words.length;
  checkInvariants();
}
...
public static BitSet valueOf(long[] longs) {
  int n;
  for (n = longs.length; n > 0 && longs[n - 1] == 0; n--);
  return new BitSet(Arrays.copyOf(longs, n));
}
```

The `valueOf(long[])` method takes in an array, copies it, and stores it in the internal `words` field of a new bitset instance. The `valueOf(long[])` method does not specify any preconditions: any non-null array can thus be converted to a bitset. Issues arise when the user calls `valueOf(long[])` with an array that has a bit set beyond index `Integer.MAX_VALUE`. This is for example the case when `longs.length` is larger than 2^{25} and contains non-zero elements in that part: since longs are 64-bit, arrays with 2^{25} elements cover all $64 * 2^{25} = 2^{31}$ non-negative integer indices. Listing 8 shows an example how this can go wrong.

Listing 8. Example of how the bug can occur with `valueOf(long[])`.

```
final int MAX_WIU = Integer.MAX_VALUE/Long.SIZE + 1; // 2^25+1
BitSet normal = new BitSet();
normal.set(0);
long[] largeArray = new long[2*MAX_WIU + 1];
largeArray[largeArray.length - 1] = 1;
BitSet broken = BitSet.valueOf(largeArray);
broken.set(0);
```

The constant `MAX_WIU` equals $2^{25} + 1$ (the bound of `wordsInUse` as determined in Sect. 3.1). The `BitSet` class can only access elements of the array up to

[4] This bug report has been accepted by Oracle, see JDK-8311905.

largeArray[MAX_WIU-1]. As a result, the bit set at largeArray[2*MAX_WIU] is *not* accessible from the broken instance(!)

The equals(Object obj) method specifies that two bitsets are equal "if and only if ... for every non-negative int index k, ((BitSet)obj).get(k) == this.get(k) [is] true." [1] However, this is not the case here: the equals() method returns false when comparing normal to broken, yet normal.get(k) equals broken.get(k) for every non-negative integer k. Furthermore, the length() method says both objects have the same logical length 1.

Going back to the resulting value from length() of broken: in this case, the return value did not only overflow to Integer.MIN_VALUE, but has even gone back up to 1. So broken and normal have the same length as observed through length(). This problem is not limited to only this example. An array with length 4*MAX_WIU+1 for which the last word is set to 1 will result in the same length() value, but in this case the length() has wrapped around *twice*.

Listing 9. The length() method calculates its returned value using wordsInUse.

```
public /*@ strictly_pure @*/ int length() {
  if (wordsInUse == 0) return 0;
  return BITS_PER_WORD * (wordsInUse - 1) + (BITS_PER_WORD -
         Long.numberOfLeadingZeros(words[wordsInUse - 1]));
}
```

The issue with length() persists when interacting normally with the broken bitset: if the user sets a bit $i > 0$ using broken.set(i), then the expected behaviour would be that length() would return $i + 1$. Instead it remains at 1, as the value of wordsInUse was not changed due to wordsInUse already being higher than any value (MAX_WIU or lower) that BitSet would ever normally assign to it, which means that the calculated value of length() is not affected (see Listing 9). Note that in some methods that call length() such as clear(int,int) and previousSetBit(int) behaviour is not negatively affected, for the same reason, that wordsInUse is already higher than expected.

This issue in the valueOf(...) methods does not appear to be a mistake in its implementation. In fact, based on the specification of the methods, a user could use the class to for example convert a LongBuffer to a long array: the user uses the valueOf(LongBuffer) method to get a bitset based on the LongBuffer, and then uses BitSet's toLongArray() method to then convert to a long array. The current implementation of the methods allows for this, provided that the last element of the buffer has at least one bit set (and so is not 0).

But this issue nicely demonstrates the utility of formal specifications: using the methods in this way results in BitSet objects that break crucial internal class invariants, causing public methods to malfunction.

4.3 Solution Directions

We now discuss possible solutions to the issues raised above. To structure the discussion, we distinguish between two solution *directions*: permit using the bit with index Integer.MAX_VALUE, or forbid using that bit. We show which changes are required to the specification (method contracts and class invariant) and implementation to realise these solutions.

Permit Using ***Integer.MAX_VALUE*** *bit.* Many operations on `BitSet` work fine out-of-the-box for the full range of integers, even when the bit at index `Integer.MAX_VALUE` is used. We show how the methods `get(int,int)`, `length()` and `valueOf(...)` can be fixed while allowing to use that bit.

As stated in Sect. 4.1, `length()` returns the negative value `Integer.MIN_VALUE` if the bit at index `Integer.MAX_VALUE` is set. Note however that no information is lost by returning `Integer.MIN_VALUE`: clients can distinguish bitsets in which the bit at index `Integer.MAX_VALUE` *is* set (returning `Integer.MIN_VALUE`) from `BitSet`s where the bit is not set (returning a non-negative length). Hence, a simple fix is to add to the Javadoc specification that the `length()` method "returns `Integer.MIN_VALUE` if the bit at index `Integer.MAX_VALUE` is set." Effectively, this means the client can interpret the negative return value as an unsigned 32-bit integer.

Using the above solution for `length`, we now turn to the `get(int,int)` method. Listing 1 showed that for the `get` method, the upper bound, given by the second parameter `toIndex`, is *exclusive*, so the highest bit the method can access is at index `Integer.MAX_VALUE-1`. Hence, if the `length()` overflows, we can simply pretend it returned `Integer.MAX_VALUE`. This yields the solution show in Listing 10.

Listing 10. A possible solution of the bug in `get(int,int)`.

```
...
int len = length();
if (len < 0)
  len = Integer.MAX_VALUE;
if (len <= fromIndex || fromIndex == toIndex)
  ...
```

This simple fix thus only requires a two-line code change in the internal implementation and does not affect the method specification, nor does it require changes to the class invariant.

For the `valueOf(...)` methods, the question arises what to do if an array is passed in that is too large (i.e. contains bits that are set beyond `Integer.MAX_VALUE`). An obvious fix is to simply prevent such arrays by throwing an `IllegalArgumentException`, along the lines of Listing 11. We also add the constant `MAX_WIU` to the `BitSet` class, initialising it with the value `Integer.MAX_VALUE/Long.SIZE + 1`.

Listing 11. A possible fix for `valueOf(long[] longs)` at the beginning of the method.

```
int len = longs.length;
if (len > MAX_WIU)
  throw new IllegalArgumentException("Input array length " + len +
                                     " is larger than maximum");
```

More lenient approaches (not shown here) are also possible: one can allow larger arrays, as long as all bits above the `Integer.MAX_VALUE` index are set to

0, or ignore such bits and only copy the first `Integer.MAX_VALUE` bits. In all those cases, the specification must also be updated to reflect these changes.

Forbid Using the ***`Integer.MAX_VALUE`*** *bit.* The second solution direction is to systematically forbid access to the bit with index `Integer.MAX_VALUE`. This can be enforced in the code by throwing an exception in methods with index parameters, along the lines of Listing 12.

Listing 12. Preventing access to the `Integer.MAX_VALUE` bit.

```
if (bitIndex == Integer.MAX_VALUE)
  throw new IndexOutOfBoundsException("bitIndex " + bitIndex +
                     "must be smaller than " + Integer.MAX_VALUE);
    ...
```

Now, the length cannot overflow, so the implementation of the `length()` method and `get(int,int)` method do not have to be changed. The `valueOf` method can be fixed along the lines of the above solution, but with an additional check to ensure that the `Integer.MAX_VALUE` bit is not set. Furthermore, it enables the methods with `fromIndex` (inclusive) and `toIndex` (exclusive) parameters, such as the `get(int,int)` method, to access all bits of a `BitSet`: since the highest bit has index `Integer.MAX_VALUE-1`, it can be accessed by taking `Integer.MAX_VALUE` for `toIndex`. The class invariant can also be strengthened to take into account that the `Integer.MAX_VALUE` bit cannot be used.

Discussion. We now briefly reflect and compare the two solution directions. The first direction enables using the full range of non-negative integer indices. It requires few and relatively small changes: the specification of `length()` is strengthened, the specification and implementation of `valueOf` is changed and the internal implementation of `get(int,int)` is fixed. This does not break existing clients that acted in good faith: `length` behaves the same, but its behaviour is now guaranteed in the Javadoc specification. The behaviour of `valueOf` is not changed when arrays are passed in with at most `Integer.MAX_VALUE` bits. But, bad faith clients that relied on the presence of these bugs (e.g. by passing an array to `valueOf` that is too large) cannot do so anymore. On the negative side, the methods with two index parameters where the upper bound is exclusive cannot access the `Integer.MAX_VALUE` bit.

The second solution direction forbids using the `Integer.MAX_VALUE` bit. It requires changing many implementations and specifications, except methods such as `get(int,int)`: all methods with a single index parameter are affected and may now throw an exception. This may break existing client code that relies on the full range of integer indices. On the positive side, the methods with two index parameters can now access the same set of bits in a `BitSet` as their single index parameter counterparts.[5]

[5] This solution direction was also considered in an issue from 2003 with `nextClearBit(..)`, see JDK-4816253, but the bugs we described above were not discovered.

5 Towards Formal Verification of the BitSet Class

One reason why formal verification of real-world software is costly is that software changes. We reported the above issues to the Java developers (including a suggested fix for the `get(int,int)` method)[6]. This discussion is ongoing at the time of writing and it is not yet clear how the `BitSet` class will be fixed. In particular, the specification and implementation of `BitSet` is not settled yet. Hence, this section is speculative, since the Java developers ultimately are responsible for choosing which solution direction to take to solve the issues mentioned above.

Instead, we will informally describe how the proof of `get(int,int)` can be carried out (Sect. 5.2), assuming that the issues described above are resolved in one particular way (discussed in Sect. 5.1). Moreover, we experienced some issues with the KeY theorem prover (see Sect. 5.3), which block us from completing the formal proof.

5.1 Background

As explained in Sect. 3, we write our formal specification in JML, which is translated into JavaDL by KeY. We add the bounds as described in Sect. 3.1 to the class invariant (see Listing 13). Furthermore, we add a condition that indicates to the KeY prover that each element in the `words` array is within the integer bounds of the primitive `long` type, and we use a KeY-specific extension of JML to do so, the so-called `\dl_` escape hatch [4]. To be able to apply various taclets that are sound only for primitive longs, we require the assumption that each array element of words satisfies the `inLong` predicate. However, we did not manage to automatically show this in KeY itself, even though the type information of the `words` array is known to KeY.

Listing 13. The last part of the class invariant, continuing Listing 2.

```
/*@ invariant
  @ ... &&
  @ (wordsInUse<=Integer.MAX_VALUE/BITS_PER_WORD+1) &&
  @ (words.length<=2*(Integer.MAX_VALUE/BITS_PER_WORD+1)) &&
  @ (\forall \bigint i; 0 <= i < words.length; \dl_inLong(words[i]));
  @*/
```

We have used the KeY theorem prover version 2.10.0.

5.2 Proof Sketch of get(int,int)

In this exposition we will sketch out the proof of correctness of the `get(int,int)` method. For the purposes of this explanation, we assume the bug is fixed according to our suggested fix permitting the `Integer.MAX_VALUE` bit. The full method body is visible in Listing 14.

---

[6] See https://github.com/openjdk/jdk/pull/13388.

Listing 14. The full method body of the `get(int,int)` method, including our suggested fix and our loop invariant.

```
public BitSet get(int fromIndex, int toIndex) {
    checkRange(fromIndex, toIndex);
    checkInvariants();

    int len = length();
    if (len < 0) // Our proposed bug fix
        len = Integer.MAX_VALUE;

    // If no set bits in range return empty bitset
    if (len <= fromIndex || fromIndex == toIndex)
        return new BitSet(0);
    if (toIndex > len) // An optimization
        toIndex = len;

    BitSet result = new BitSet(toIndex - fromIndex);
    int targetWords = wordIndex(toIndex - fromIndex - 1) + 1;
    int sourceIndex = wordIndex(fromIndex);
    boolean wordAligned = ((fromIndex & BIT_INDEX_MASK) == 0);

    // Process all words but the last word
    /*@ // Adjusting wordsToSeq for result:
      @ maintaining (\forall \bigint j;
      @   0 <= j < ((\bigint)i*(\bigint)BITS_PER_WORD);
      @    ( (result.words[j / BITS_PER_WORD]
      @          >>> (int)(j % BITS_PER_WORD)) & 1 )
      @  == (fromIndex + i < wordsToSeq().length
      @            ? wordsToSeq()[fromIndex + i] : 0) );
      @   // >>> is not defined for bigint.
      @ maintaining i >= 0 & i <= targetWords - 1;
      @ maintaining sourceIndex < wordsInUse;
      @ maintaining (i < targetWords-1)
              ==> sourceIndex+1 < wordsInUse;
      @ maintaining sourceIndex >= fromIndex / 64 &&
                     sourceIndex <= toIndex / 64;
      @ maintaining (\forall \bigint j; 0 <= j < result.words.length;
         \dl_inLong(result.words[j]) );
      @ assignable result.words[*];
      @ decreasing targetWords - i;
      @*/
    for (int i = 0; i < targetWords - 1; i++, sourceIndex++)
        result.words[i] = wordAligned ? words[sourceIndex] :
            (words[sourceIndex] >>> fromIndex) |
            (words[sourceIndex+1] << -fromIndex);

    // Process the last word
    long lastWordMask = WORD_MASK >>> -toIndex;
    result.words[targetWords - 1] =
        ((toIndex-1) & BIT_INDEX_MASK) < (fromIndex & BIT_INDEX_MASK)
        ? /* straddles source words */
        ((words[sourceIndex] >>> fromIndex) |
         (words[sourceIndex+1] & lastWordMask) << -fromIndex)
        :
        ((words[sourceIndex] & lastWordMask) >>> fromIndex);

    // Set wordsInUse correctly
    result.wordsInUse = targetWords;
    result.recalculateWordsInUse();
    result.checkInvariants();

    return result;
}
```

Initialising Local Variables. After input validation, the `get` method calls several small methods that do not modify any fields of pre-existing objects. These methods have all been given contracts, the main one being `wordIndex(i)`, which returns $i/64$ for non-negative `i`. Besides `length()`, these contracts have all been verified either automatically or with minimal human interaction in KeY.

Next, several local variables are initialised in lines 15-18. First, a bitset `result` is created through a public constructor, with a `words` array that can fit all the bits required, and `result.wordsInUse` is initialised to 0. The `words` array is filled directly. `result.wordsInUse` is only updated after it is filled completely. The integer `targetWords` is the number of words to copy to `results.words`, and has the same value as `results.words.length`. The `sourceIndex` variable indicates the starting index in `this.words` of the bits to copy. The boolean `wordAligned` indicates if the `result` bitset is aligned to the original bitset. If this is *not* the case, then copying the bits is made more complicated, as each element of `result.words` is spread across two elements of `this.words`.

Loop Invariant. The clause of the loop invariant on line 22 is an adjusted version of `wordsToSeq()`. As `result.wordsInUse` is 0 during the loop, we cannot use `wordsToSeq()` to track the copied bits in `result.words`, as it has a zero length when `wordsInUse` is zero. So, the loop counter `i` takes care of this.

To verify the statements from line 29 onwards, we use a number of lemmas. First, the number of words that the method copies (`targetWords`) is less than or equal to the number of logically defined elements of `words` (`wordsInUse`). The largest value `toIndex` can have is `wordsInUse`*64, as the `get(int,int)` method reduces `toIndex` so that it is within the logically significant length of the `BitSet`. Hence, the largest value `targetWords` can have is `wordsInUse`, in the case of $\frac{toIndex - fromIndex - 1}{64} + 1 = \frac{wordsInUse * 64 - 0 - 1}{64} + 1 \leq wordsInUse$.[7]

Using this bound for `targetWords`, we can verify that in the loop body, the expressions `this.words[sourceIndex]` and `this.words[sourceIndex+1]` have significant bits as bounded by `wordsInUse`. This is needed for establishing the relation between the resulting bitset and the current bitset, and for preventing an exception.

$$sourceIndex + targetWords - 1 < wordsInUse.$$

This can be rewritten to:[8]

$$\frac{fromIndex}{64} + \left(\frac{toIndex - fromIndex - 1}{64} + 1\right) - 1 < wordsInUse.$$

Next, consider division of *fromIndex* by 64: we can write $fromIndex = 64k + x$ with $k \geq 0$ and $0 \leq x < 64$. Plugging this in into the above equation we can derive that the left-hand side equals $(64k + x)/64 + (toIndex - 1 - x - 64k)/64$. By Java's integer division semantics (where non-negative results are rounded down),

[7] Rounded using Java rules.

[8] Note that both `sourceIndex` and `targetWords` are calculated using `wordIndex(...)`.

this equals $k + (toIndex - 1 - x)/64 - k = (toIndex - 1 - x)/64$. Clearly this is smaller or equal to $(toIndex - 1)/64$. This is smaller than `wordsInUse`, using the bound for `targetWords` proved before, so the desired inequality follows.

Finally, if `((toIndex-1) & BIT_INDEX_MASK) < (fromIndex & BIT_INDEX_MASK)` holds[9], then the boolean `wordAligned` must be false (as f `(fromIndex & BIT_INDEX_MASK)` must be larger than 0), and we know that the method uses `sourceIndex+1` to access the `this.words` array. To compensate for the `+1`, we set the bound of `sourceIndex+targetWords` to `wordsInUse-1`. The proof for this is similar to the previous inequality.

As KeY does not fully support binary AND operations (see Sect. 5.3), we replaced `n & 63` with `n % 64`. These are equivalent for non-negative `n`. With suitable lemmas, we expect the preservation of the loop invariant is provable.

End of the `get(int, int)` *Method.* Once all bits have been copied from the original bitset to `result`, the method calls the `recalculateWordsInUse()` method to establish the invariant in `result`. In our case, `wordsInUse == 0 || words[wordsInUse - 1] != 0` and `wordsInUse == words.length || words[wordsInUse] == 0` from the class invariant need not be true when the method starts (the method is in fact responsible for re-establishing these properties). In particular, `wordsInUse` may be too high, so `words[wordsInUse-1]` may be zero. All other clauses from the class invariant do hold initially. To restore the class invariant, the method lowers `wordsInUse` to the most significant element of `result.words` that is not zero (and to zero if there is none).

As can be seen above, a substantial part of both the (development of) the specification and the proof concerns dealing with Java's bounded integer semantics. We now reflect briefly on our approach, where we chose to deal with Java's bounded integer semantics right from the start. The question may arise whether a two-step approach would have been simpler, where as a first step, a proof of the `BitSet` class is given using ordinary mathematical integer semantics and in a second step, this proof is amended by using Java's bounded integer semantics. KeY supports both mathematical integer semantics and Java's bounded semantics so on first sight a two-step approach may sound promising.

But consider our bug fix in Listing 14, line 6. Without overflows, `length()` returns a positive integer, so the true-branch is dead code with mathematical integer semantics. In the bounded integer version it is not dead code and causes execution to proceed differently in the subsequent code. Formally, the program using bounded integer semantics is not a refinement of that 'same' program with mathematical integers: it satisfies different properties/contracts. Different specifications may have to be developed for the two different integer semantics, symbolic execution of the method proceeds rather differently as witnessed by the dead-code example above and consequently, different proof obligations are generated (which in turn requires different proofs). This complicates proof reuse between the two 'steps'. Practically, the division into two steps would thus amount to an extra step where one would investigate which specifications,

[9] BIT_INDEX_MASK is a constant integer equalling 63.

proofs etc. would be needed for the non-real-world version that uses mathematical integers. We chose to avoid such an extra step and deal with the Java's actual bounded integer semantics from the beginning.

5.3 Required Extensions to KeY

Bit shift operations, such as the >>> and << used in `get(int,int)`, cause the so-called *Finish symbolic execution* macro to get stuck in a loop, endlessly applying rules on the shift term. There are workarounds, such as by hiding the shift terms, but this comes at the cost of more manual interactions.

More importantly, KeY currently lacks full support for bitwise operators, such as `binaryOr` and `binaryAnd`, which prevents a full mechanic verification of the class. Rules need to be added, or the terms could be translated to an SMT solver, which could then handle these bitwise operations. It may be possible to develop a general theory involving `binaryOr` and `binaryAnd` operators, but in our case this does not appear to be necessary. A large amount of the proof goals (not discussed here) are related to `wordsToSeq()`. An individual element of this sequence is a single bit. This knowledge can be used to make rules where one or both of the operators are a single bit, allowing us to add specific, but simple rules to KeY. Listing 15 shows an example of such a rule for the `binaryOr` operation.

Listing 15. Taclet rule for binaryOr.

```
// x | y = 0. This is true iff x = 0 and y = 0.
orLongZero {
  \schemaVar \term int x, y;
  \assumes(inLong(x), inLong(y) ==>)
  \find(moduloLong(binaryOr(x, y)) = 0)
  \sameUpdateLevel
  \replacewith(x = 0 & y = 0)
};
```

This rule is necessary to close the proof of `BitSet`'s `set(int)` method. In general, all specific rules we *need* should follow from a more general theory involving bitwise operators. But since the implementation and Javadoc specification are not settled yet, the precise rules that are needed are not known yet, so we left development of the proof rules as future work.

6 Conclusion

We discussed OpenJDK's `BitSet` class, formulated its formal specification and wrote tests. Using these formal analyses, we discovered bugs triggered by integer overflows and proposed several solution directions for resolving these issues. The integer overflow in `length()`'s return value when the `Integer.MAX_VALUE` bit is set is a relatively minor issue, as the method is still usable as long as the user takes this possibility into account. Meanwhile, the bug discovered in the `get(int,int)` method prevents the method from being properly functioning as long as the `Integer.MAX_VALUE` bit is set. The bug in the `valueOf(..)` methods allows the user to create objects which contain inaccessible bits. The `length()`

method is no longer reliable in these objects due to an integer overflow. Both of these bugs are significant, as they fundamentally break the (intended) specification of the `BitSet` class. Finally, we discussed initial steps towards verification of the `get(int,int)` method and illustrated remaining challenges. The artifact with formal specifications and proofs for several smaller methods is publicly available at [16].

References

1. BitSet (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html. Accessed 12 May 2023
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice. LNCS, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
3. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular verification of JML contracts using bounded model checking. In: Margaria, T., Steffen, B. (eds.) ISoLA 2020. LNCS, vol. 12476, pp. 60–80. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-61362-4_4
4. Bian, J., Hiep, H.A., de Boer, F.S., de Gouw, S.: Integrating ADTs in KeY and their application to history-based reasoning about collection. Formal Methods in System Design, pp. 1–27 (2023). https://doi.org/10.1007/s10703-023-00426-x
5. Blot, A., Dagand, P.É., Lawall, J.: From sets to bits in Coq. In: Kiselyov, O., King, A. (eds.) FLOPS 2016. LNCS, vol. 9613, pp. 12–28. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-29604-3_2
6. de Boer, M., de Gouw, S., Klamroth, J., Jung, C., Ulbrich, M., Weigl, A.: Formal Specification and Verification of JDK's Identity Hash Map Implementation. In: ter Beek, M.H., Monahan, R. (eds.) Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7–10, 2022, Proceedings. LNCS, vol. 13274, pp. 45–62. Springer (2022). https://doi.org/10.1007/978-3-031-07727-2_4
7. Böhme, S., Fox, A.C.J., Sewell, T., Weber, T.: Reconstruction of Z3's bit-vector proofs in HOL4 and Isabelle/HOL. In: Jouannaud, J.-P., Shao, Z. (eds.) CPP 2011. LNCS, vol. 7086, pp. 183–198. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-25379-9_15
8. Cheon, Y., Leavens, G., Sitaraman, M., Edwards, S.: Model variables: cleanly supporting abstraction in design by contract: research articles. Softw. Pract. Exper. **35**(6), 583–599 (2005)
9. Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., Trtik, M.: JBMC: a bounded model checking tool for verifying Java bytecode. In: Chockler, H., Weissenbacher, G. (eds.) CAV 2018. LNCS, vol. 10981, pp. 183–190. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96145-3_10
10. De Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In: Computer Aided Verification: 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18–24, 2015, Proceedings, Part I 27. pp. 273–289. Springer (2015)
11. Hadarean, L., Barrett, C., Reynolds, A., Tinelli, C., Deters, M.: Fine grained SMT proofs for the theory of fixed-width bit-vectors. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) LPAR 2015. LNCS, vol. 9450, pp. 340–355. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48899-7_24

12. Hiep, H.-D.A., Bian, J., de Boer, F.S., de Gouw, S.: A tutorial on verifying `LinkedList` using KeY. In: Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M. (eds.) Deductive Software Verification: Future Perspectives. LNCS, vol. 12345, pp. 221–245. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64354-6_9
13. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY (extended paper). Int. J. Softw. Tools Technol. Transf. **24**(5), 783–802 (2022). https://doi.org/10.1007/s10009-022-00679-7
14. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A Notation for Detailed Design (1999). https://doi.org/10.1007/978-1-4615-5229-1_12
15. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
16. Tatman, A.S., Hiep, H.A., de Gouw, S.: Analysis and Formal Specification of OpenJDK's BitSet: Proof Files (2023). https://doi.org/10.5281/zenodo.8043379
17. Zohar, Y., Irfan, A., Mann, M., Niemetz, A., Nötzli, A., Preiner, M., Reynolds, A., Barrett, C., Tinelli, C.: Bit-precise reasoning via int-blasting. In: Finkbeiner, B., Wies, T. (eds.) VMCAI 2022. LNCS, vol. 13182, pp. 496–518. Springer, Cham (2022). https://doi.org/10.1007/978-3-030-94583-1_24