Opleiding Informatica

Formal Specification and Analysis

of OpenJDK's `BitSet` Class

Andy S. Tatman
S2946114

Supervisors:
Prof. dr. Marcello Bonsangue
drs. Hans-Dieter A. Hiep

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

25/07/2023

Abstract

In this thesis, we use a combination of formal specification, testing and formal verification techniques to analyse OpenJDK's `BitSet` class. This class stores a bit vector that grows when required. We write a formal specification for the class in Java Modelling Language (JML), with the intention of using the KeY theorem prover to formally verify the correctness of the class. During our analysis, we discovered a number of bugs in the code. We describe how these bugs occur using our formal specification. We then set out different possible solution directions for these issues, and discuss the advantages and disadvantages of each. We discuss why we chose KeY as our verification tool, and detail extensions that KeY requires in order to be able to verify the correctness of the `BitSet` class. We provide some rules we have created, and use these and our formal specification to verify `BitSet`'s `set(int)` method. We also discuss some of the proof steps required to verify the correctness of the `get(int,int)` method, once the bugs we discuss have been fixed.

Contents

1 Introduction

An essential part of software development is ensuring that the software we have created is 'correct'. This can include making sure it does what it is meant to do, but also that it does not have any unintended behaviour such as crashes or security issues. There are many different ways of checking software, such as by using a debugger or by writing unit tests. These techniques will succeed in finding many bugs in the software, but they do not guarantee that the software is actually correct. In order to guarantee software's correctness, we can instead use formal specification and formal verification.

Formal specification involves writing exactly what a piece of code does and does not do, while informal specifications (especially if they are only written in words) can sometimes be unclear or contain contradictions. In order to do so unambiguously, formal specifications are often written with a similar syntax to the code itself, but using logical and mathematical operators. As an example, you might specify that 'for all integers `x` and `y`, the method `sum(x,y)` will return the value `x+y`'. In programming languages such as Java, there is no 'for all integer ...' construct, as this would involve checking every possible value. Meanwhile, the $\forall$ operator is common-place in logical languages. An advantage of formalising the programme's specification, is that it forces you as for example a programmer or as someone analysing the programme to determine exactly what the programme does, and what it does not do.

Formal verification takes the formal specification and checks that the code correctly satisfies the behaviour that the specification expects. The process of formal verification involves going through every possible execution path of the programme (a process known as symbolic execution) and determining whether the result of the programme matches up to the expected results in the specification. This is done in the form of a formal proof, using proof rules to show that each path and each proof case is correct.

Formal specification combined with formal verification is a more effective debugging technique compared to for example unit tests, in the sense that a completed verification means you can guarantee that execution a piece of code complies to the specified behaviour, meaning that no bugs can occur which would break the specification. If a bug does exist, then you may find this in the process of making the formal specification or while trying to verify the specification. A bug means that the specified behaviour does not occur for legal instances in the specification. As a result, it will be impossible to verify that the program satisfies its specification, which is a good indicator that either the programme or the specification contains an error.

The traditional process of developing code involves (informally) specifying the code, followed by a cycle of writing the code and testing/debugging the code. Usually, the testing/debugging go hand in hand with writing the code. If you have discovered a bug using a unit test, you can change the code to fix the bug and then immediately check that the issue is fixed with the unit test. While testing/debugging techniques can be used to *find* bugs, they cannot be used to *prove* that no bugs exist.

Using formal techniques, you may also initially have a similar process of writing and testing/debugging. When you believe the code may be correct, you can then start drawing up a formal specification, and then using this specification to carry out the verification. Unfortunately, formal specification and verification are also more time-consuming than other techniques for ensuring code quality. If the code is changed at all, e.g. because you discovered a bug, then the verification process needs to be restarted, as the existing proof will not apply to the new code. A verification proof of even a small section of code may require several thousand proof steps. While much of this can be automated using proof assistants, this will still require intense effort from the user in order to complete the verification.

Because it is so time-consuming, formal specification and verification efforts are generally reserved for code that is either essential or very frequently used, such as standard library code. In this thesis, we discuss and analyse OpenJDK's `BitSet` class, which is part of in Java's standard library. To our knowledge, this is the first time the source code of OpenJDK's `BitSet` class is analysed using formal specification and verification. While writing this thesis, we discovered that part of `BitSet`'s class has previously been specified prior to OpenJDK being released [Lea02]. However, that specification appears to have been created without access to the source code, and thus does not use the actual fields of the class. As a result, the previous analysis did not lead to the discoveries we have made.

The original plan for this research was to formally specify and verify the correctness of a number of `BitSet`'s methods. However, during our analysis we discovered a number of bugs in the class, which appear to have been present in the class since OpenJDK first became public in 2007[1]. We first identified a bug in the `BitSet`'s `get(int,int)` method. Later on, we also discovered an issue with the specification of the class' static `valueOf(..)` methods, which results in bitset objects that do not behave as expected.

For the formal verification, we use the KeY theorem prover. KeY is a proof assistant that can take Java code annotated with a JML formal specification as input, and can perform formal verification through symbolic execution. It is currently the best option for the formal verification of Java programmes, due to its ease of use (it is designed to work directly on Java code) and because it accurately models Java semantics, such as integer overflows. However, we have found that KeY requires extensions before it can verify the `BitSet` class.

In Section 2 we will discuss prior work related to this topic. In Section 3 we will introduce the `BitSet` class, including a number of its methods and an example use case, and discuss the existing informal specification. In Section 4 we will introduce Java Modelling Language (JML), the language we use to write our formal specification. We then write a formal specification for the class and a selection of its methods in JML. Section 5 discusses the two bugs we discovered in the class. We then offer two main directions towards solving the various issues we discovered, and discuss each direction's advantages and disadvantages. Section 6 introduces KeY, and explains why it requires extensions in order to complete verification of the `BitSet` class. We provide some of these required extensions, and explain how we use them to verify the correctness of the `set(int)` method, followed finally by a partial sketch of a proof of the `get(int,int)` method.

This thesis was written as part of the Informatica (Computer Science) bachelor at Leiden University, and was supervised by Prof. dr. Marcello Bonsangue and drs. Hans-Dieter A. Hiep. Together with drs. Hiep and dr. Stijn de Gouw, this research also resulted in an article which we have submitted to a conference.

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

2 Related Work

The Java Modelling Language (JML) is a language which extends the regular Java language and allows users to write formal specifications for Java code [LBR99]. JML statements are written in the comments of the Java code, and as such does not affect the behaviour of the code. One important aspect of JML that we use in this paper is the ability to define model methods, which exist outside of the actual programme and allow us to simplify the specification [CLSE05].
One way of using JML is through tools such as the KeY theorem prover to perform static verification of the programme to ensure it satisfies its specification. The most useful resource for working with the KeY theorem prover, is the KeY book [ABB+16]. Other papers have used JML and KeY to analyse other parts of Java, and have also identified issues. Examples of this include an analysis of OpenJDK's `LinkedList` class [HBdBdG20] and of Java's `BigInteger` class [Pfe17], as well as various sorting algorithms implemented in Java [ABB+16, DGRdB+15].
While writing this thesis, we discovered that some previous work has been done in specifying the `BitSet` class [Lea02]. We analyse this specification after introducing the notation and presenting our own, so that we can make a comparison. See Section 4.9. We will also discuss why this previous specification does not apply to the current version of the `BitSet` class.

In Section 6, we discuss issues KeY has with bitwise operators. Other theorem provers have similarly worked on implementing bitwise operators. In the Coq proof assistant, a library has been implemented that converts bitsets (as used in programming) to a finite set (as used by Coq in proofs) [BDL16]. In the Isabelle proof assistant, a generalised collection of theories has been developed to reason over machine words such as integers of arbitrary length [Daw09]. An alternative technique for handling bitwise operators using SMT solvers (not discussed here) is called 'bit-blasting', and is discussed in [Kro09]. Using this technique, our 64 bit integers may be converted to a CNF formula, which can then solved using an SMT solver.

3 The **BitSet** class

The `BitSet` class is a standard library class of the Java language. It has been made open-source through the Open Java Development Kit (OpenJDK). This is an open source implementation of the Java standard library, and was released by the developers of Java back in 2007. The class allows users to store bits in a bit vector, packing these bits into an array of type `long`. This is more efficient than using an array of booleans, as the size of an individual boolean (and therefore by extension an array of booleans) in Java is not "precisely defined" [jav].
Listing 1 showcases the various fields and methods of this class that are relevant to this thesis. The `/** ... */` comments written above the public methods are cited from Java's specification of the `BitSet` class [Bit].

Listing 1: Fields and methods of the `BitSet` class relevant to this thesis.

```
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;

```

```java
  /** 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) { ... }

  // Small methods used by a number of other methods:
  // Returns (bitIndex / 64) if bitIndex >= 0, or -1 if bitIndex = -1.
  private static int wordIndex(int bitIndex) { .. }
  // Checks the invariant.
  private void checkInvariants() { .. }

  /** Sets the bit at the specified index to true. */
  public void set(int bitIndex) { ... }
  // Methods used by the set(int) method:
  // Ensures that wordsInUse and words.length are both >= wordIndex+1.
  private void expandTo(int wordIndex) { .. } // Helper method
  // Ensures that words.length >= wordsRequired.
  private void ensureCapacity(int wordsRequired) { .. }

  /** Sets all of the bits in this BitSet to false. */
  public void clear() { ... }

  /** Returns the value of the bit with the specified index. */
  public boolean get(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) { ... }
  // Method used by the get(int,int) method:
  // Lowers the value of wordsInUse, in order to ensure that the invariant holds after termination.
  private void recalculateWordsInUse() { .. } // Helper method

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

  /** Compares this object against the specified object. */
  public boolean equals(Object obj) { .. }
}
```

We let **false** stand for the bit value 0, and **true** for the bit value 1.

The class stores its bits in an array of 64 bit long elements, called the `words` array. Every bit in a word is used to pack bits, including the sign bit. Index 0 is the least significant bit of the first word, and index 63 is the most significant bit of the first word (the sign bit).

0-63	64-127	128-191
1	0	2305843009213693952

Figure 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.

0	1	2		189	190	191
1	0	0	···	1	0	0

Figure 2: The logical representation of the same bitset as depicted in Figure 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**.

Figure 1 shows an example of the **words** array for a bitset instance, while Figure 2 shows the logical representation of this same bitset. The class uses an integer **wordsInUse** to keep track of the last word that contains at least one bit set to 1. The class uses **wordsInUse** to approximate the logical length of the bitset, such as when calculating the value of **length()**.
The logical length of `BitSet` is defined by the last position of the most significant bit set to 1. This most significant bit is stored in **words[wordsInUse-1]**. In the example above, the logical length is 190, as 189 is the last bit that is set to 1.
Initially every bit in a bitset is set to 0. If the user tries to retrieve the value of a bit with an index outside of the logical length of a bitset, then this value is 0 by default. This allows the class to handle accesses to any bit on a non-negative position, even if its index falls outside the actual **words** array. When the user sets a bit outside of the logical length of a bitset, then the bitset is expanded to ensure the index fits in the new logical length.

Informal specification of relevant public methods

- **length()**: Returns the position of the most significant bit set to 1, plus 1. If **length()** returns a value > 0, then the bit at position **length()**-1 is set to 1. All bits at positions strictly greater than **length()**-1 have the value 0.

- **void set(int bitIndex)**: If **bitIndex** is non-negative ($0 \leq$ **bitIndex**), then the bit at position **bitIndex** in the bitset is set to true. If **bitIndex** is larger than or equal to the logical length of the bitset (**length()**), then the bitset expands in order to fit the **bitIndex** bit. The new value of **length()** is **bitIndex+1**.

- **void clear()**: The method sets every bit in the bitset to false. The value of **length()** becomes 0.

- **boolean get(int bitIndex)**: If **bitIndex** is non-negative ($0 \leq$ **bitIndex**), then the method returns the value of the bit stored at position **bitIndex** in the bitset. If **bitIndex** is larger than or equal to the logical length of the bitset (**length()**), then the method will always return false.

- `BitSet get(int fromIndex, int toIndex)`: `fromIndex` must be greater than or equal to 0 and `toIndex` must be greater than or equal to `fromIndex` ($0 \leq$ `fromIndex` $\leq$ `toIndex`). If this is the case, then the method will return a new bitset, where the bits from `fromIndex` up to but *not* including `toIndex` have been copied.
  The bit at `fromIndex` in the *original* bitset is at position 0 in the *new* bitset, position 1 in the *new* bitset equals the bit at `fromIndex`+1, etc..., until the bit at position `toIndex-1` in the *original* bitset which is stored at position `toIndex-1-fromIndex` in the *new* bitset.

Example use of the `BitSet` class

We provide a small example of the class being used in Listing 2.

Listing 2: An example of the `BitSet` class being used.

```
1  BitSet bset = new BitSet(64);
2  boolean value = bset.get(10); // value = false.
3  int len = bset.length(); // len = 0.
4  bset.set(10);
5  value = bset.get(10); // value = true.
6  len = bset.length(); // len = 11.
7
8  BitSet secondBSet = bset.get(10, 20);
9  value = secondBSet.get(10); // value = false.
10 value = secondBSet.get(0); // value = true.
11 len = secondBSet.length(); // len = 1.
12
13 bset.clear();
14 value = bset.get(10); // value = false.
15 len = bset.length(); // len = 0.
```

On line 1 we create an empty bitset. At this point, all the bits are set to 0. `bset.get(i)` will return false for any integer `i` ≥ 0, including for `i = 10` (line 2). Because no bits are set, the `length()` method call on line 3 will return 0. On line 4, we set the bit at position 10. At this point, this bit is now set to true (line 5), and the `length()` method indicates that the logical length is now one higher than 10 (line 6).
On line 8, we create a new bitset using the `get(int,int)` method. We take a portion of the `bset` bitset. We take the bit that is set to true in `bset` as the first bit, followed by a number of bits that have not been set in the original bitset. (These bits are all 0, as is standard with `BitSet`.) The bit that was set in `bset` (on position 10 in `bset`) is set in `secondBSet` on position 0 (line 10). Calling `secondBSet.get(10)` (line 9) therefore returns false, as this bit is *not* set in the new bitset. The length of the new bitset is 1, as only index 0 contains a bit that is set to true (line 11).
Finally, on line 13 we call the `clear()` method, which sets every bit in the bitset to false. As a result, `bset.get(10)` once again returns false (line 14), and `bset.length()` once again returns 0 (line 15). The `bset.clear()` call has no effect on `secondBSet`, as the objects do not refer to each other. `secondBSet` is therefore unchanged.

4 Formal Specification

Our formal specification is focused on the public methods discussed previously, specifically `set(int)`, `clear()`, and `get(int,int)`, as well as smaller methods that these methods call.

When writing our specification for the public methods, we want to create a layer of abstraction between the specification and implementation. If the implementation is changed, but the method has the same purpose, then the contract should still apply to the new implementation. As an example, say the `BitSet` class is entirely rewritten. As long as the specifications of the methods and of the class do not change, then a user of the class should not notice any change. The more abstract the specification is compared to the implementation, the easier it should be for users to understand the *purpose* of the method, without having to worry over *how* the method achieves this purpose.

We will write the method contracts based on the expected behaviour, which we determine by looking at Java's informal specification of the methods as well as the code and comments of the method itself. In order to work with the logical representation of the bitset, we will introduce the `wordsToSeq()` model method in Section 4.3. If the implementation of the `BitSet` class itself changes significantly, then it might mean that the model method may also need to change, but the method contracts are written in such a way that they should not need to change.

4.1 Introduction to Java Modelling Language

Java's documentation already gives us a specification of the `BitSet`'s public methods [2]. However, this is an informal specification, and is not always as specific as we may want it to be. As an example, the specification for the `set(int bitIndex)` is the following: "Sets the bit at the specified index to true." [Bit] While a human can reasonably infer that this means that other bits are not changed, the documentation does not explicitly state this.

For our formal specification, we want to be able to describe *exactly* what does and does not happen. For this, we use Java Modelling Language (JML) [LBR99]. JML is an extension of Java, and allows us to write annotations, such as contracts or assertions, in the comments of Java.

We write method contracts to formally specify what a method does. We use Listing 3 as a simple example:

Listing 3: An example of a simple contract.

```
/*@
  @ public normal_behaviour
  @  requires true;
  @  ensures (a >= b) ==> \result == a;
  @  ensures (a < b) ==> \result == b;
  @  assignable \strictly_nothing;
  @  // no_state  // for future use
  @*/
public static int max(int a, int b) { .. }
```

When making a method contract, we deal with two different states: the state *before* the method was called, and the state *after* the method terminates. The pre-condition describes what must be true before the method, and the post-condition describes what must be true after.

The pre-condition of a contract is defined using `requires` clauses. We assume these are true when the method is called. In this case, the method can be called in *any* state, as `true` holds in *any* state.

The post-condition of a contract is defined using `ensures` clauses. When the method terminates, these should be true. In this case, when the method terminates, the resulting value (`\result`) should equal the largest value between `a` and `b`.

[2] https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html

In order to compare the pre-condition and post-condition, JML introduces the `\old(..)` clause. As an example: for an integer `x`, `ensures x == \old(x);` says that the value of `x` should not have changed during the execution of that method.
The `normal_behaviour` term indicates that, given the pre-condition, this method will always terminate normally and will not raise an exception.
The `assignable` clause indicates what parts of the heap can be changed. In this case, `assignable \strictly_nothing;` tells us that nothing can be created or changed on the heap. One alternative is `assignable \nothing;`. Here, nothing that existed before has been changed, but we can make new objects. This is used for example in constructors, where nothing pre-existing is changed, but the constructor may make new objects as it initialises the class object. Another option is specifying specific objects or variables that can be changed by the method, using `assignable x,y,z;` for some objects `x,y,z`. If a method is denoted with `/*@ strictly_pure @*/`, then the method never alters the heap of the programme. This is equivalent to every contract for this method having `assignable \strictly_nothing;`.

Aside from method-specific contracts, there are also assertions that are true in both the pre- and post-condition of (nearly) every method of the class. Instead of specifying these assertions in each individual contract, these can instead be specified in the class invariant. These are implicitly added to the contracts: the invariant is assumed to be true at the start of the method, and at termination of the method the invariant must once again be true. It is possible that the method breaks the invariant during its execution, as long as the invariant is reinstated before termination.
The exception to this is a *helper* method: A method where the contract contains the **`helper`** term does not automatically have the invariant added to the pre-condition or post-condition. An example of a helper method is a private method that is called internally to restore the invariant broken by another method. In such a case, the invariant clearly will not always be true prior to calling the method, but will be on termination. Here, the method is marked as a helper, but the contract contains `ensures \invariant_for(this);` to indicate that the invariant does hold at termination.

Loops provide a special challenge to programme verification. When our code has a loop, we need to specify a loop invariant. A loop invariant is an assertion or a group of assertions which must hold at the start and end of every iteration of the loop body. As part of the proof, you need to first show that the loop invariant holds when the programme initially enters the loop. Assuming the loop invariant initially held, you then need to show that the invariant holds after executing the loop body once. This shows that the loop invariant will hold after an abstract number of iterations. We provide a simple example of such a loop invariant in Listing 4:[3]

Listing 4: An example of a simple method that uses a loop invariant.

```
/*@ public normal_behaviour
  @ requires a != null & a.length > 0;
  @ ensures (\result == false) ==> (\forall \bigint i; 0 <= i < a.length; a[i] != x);
  @ ensures (\result == true) ==> !(\forall \bigint i; 0 <= i < a.length; a[i] != x);
  @*/
public boolean find(int[] a, int x) {
```

[3]Note: This method does *not* come from the `BitSet` class. This is an example we have created for this explanation.

```
    int i = 0;

    /*@
      @ maintaining i >= 0 & i <= a.length;
      @ maintaining (\forall \bigint j; 0 <= j < i; a[j] != x);
      @ decreasing (a.length - i);
      @ assignable i;
      @*/
    while (i < a.length && a[i] != x) {
        i++;
    }

    return (i < a.length);
}
```

The method searches the array, and returns true if the integer `x` occurs in the array. The `maintaining` clauses are the main part of the loop invariant. These must hold after every iteration of the loop. The information in the loop invariant is used to detail the state both between iterations and once the loop has terminated. The `decreasing` clause tells the prover that the loop will eventually halt: every iteration, the value after `decreasing` must decrease by at least 1, and the value must always be ≥ 0. The `assignable` clause works the same way as it does for method contracts.

Finally, in Section 4.3, we will introduce a model method that we have created. A model method is a method that can only be used in JML specifications (and not in the actual Java programme), and does not affect the actual state of the programme [CLSE05]. In this case, we use it to convert the `words` array, where the bits are packed into the 64 bit longs, to a sequence of individual bits, which is our logical representation where element `i` of the sequence corresponds to bit `i` of the bitset. This also allows us to specify our contracts using the logical representation, which helps preserve our separation between specification and implementation.

4.2 Class invariant for the `BitSet` class

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

- Firstly, either `wordsInUse` is 0, or the last word in the logically defined length of `words`, i.e. `words[wordsInUse-1]`, has at least one bit that is set to 1.
- Secondly, `wordsInUse` is in the range of [0, `words.length`], inclusive.
- Finally, either `wordsInUse` equals the length of `words`, or the first word outside the logical length of the `words` array, i.e. `words[wordsInUse]`, has no set bits and so is 0.

These assertions were given in the class, and they always hold at the beginning and end of `BitSet`'s public methods. However, we have expanded the class invariant, as it can include more conditions: First of all, the `words` array is allocated in every constructor, and therefore is never null. Next, the third assertion from `checkInvariants()` appears to suggest that *every* word after `words[wordsInUse-1]` should equal 0. This is backed up by the implementation of other methods. As an example, we look at the `recalculateWordsInUse()` method. This helper method restores the invariant, by *lowering* `wordsInUse` until `(wordsInUse == 0 || words[wordsInUse - 1] != 0)` is true. Here, the method's specification assumes that every element above `words[wordsInUse - 1]` equals 0.

We combine these assertions to get the partial class invariant in Listing 5:

Listing 5: 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 want to look for potential upper bounds to `words.length` and `wordsInUse`. Bitsets generated by public constructors (i.e. not by the `valueOf(..)` methods, see Section 5.2) will allocate the `words` array. When interacting with the class using methods such as `set(..)`, the `words` array grows as required using the `expandTo(int)` and `ensureCapacity(int)` methods, while the `wordsInUse` variable is updated to reflect the largest word with a set bit. The largest position that a bit could be set to 1 is at position `Integer.MAX_VALUE`, which is stored in `words[Integer.MAX_VALUE/64]`. This means that the upper bound to `wordsInUse` is `Integer.MAX_VALUE/64 + 1`.

Listing 6: The `ensureCapacity(int)` method.

```
private void ensureCapacity(int wordsRequired) {
    if (words.length < wordsRequired) {
        // Allocate larger of doubled size or required size
        int request = Math.max(2 * words.length, wordsRequired);
        words = Arrays.copyOf(words, request);
        sizeIsSticky = false;
    }
}
```

The `ensureCapacity(int wordsRequired)` method grows the `words` array if required, specifically if `wordsRequired` is larger than the current length of `words`. As you can see in Listing 6, if the array is made longer, then the new length will be at least twice as long as the original length of the array. The bound for `wordsRequired` is the same bound as for `wordsInUse`, namely `Integer.MAX_VALUE/64 + 1`. The largest `word` array that the `BitSet` constructors can make is also `Integer.MAX_VALUE/64 + 1`. For the upper bound of the length of `words`, we take double this value, giving us an upper bound of `2*(Integer.MAX_VALUE/64 +1)`.
These bounds are maintained when using `BitSet`'s methods to interact with the bits stored. However, we will show in Section 5.2 that these do not always hold when using `BitSet`'s `valueOf(..)` methods.

4.3 The `wordsToSeq()` model method

In order to reason with the contents of a bitset, we convert the array of 64 bit elements to a logical sequence of individual booleans, such that position i in the sequence is corresponds to bit i in the bitset. This allows us to write our contracts using the logical representation, while further obscuring the actual implementation.
This conversion is done using our model method called `wordsToSeq()`, and is shown in Listing 7:

Listing 7: Our `wordsToSeq()` model method.

```
/*@ private model strictly_pure \seq wordsToSeq() {
  @  return (\seq_def \bigint i;
  @      0; (\bigint)wordsInUse*(\bigint)BITS_PER_WORD;
  @      // Note 1: Shifting is undefined for \bigint, hence why we cast (i % BITS_PER_WORD)
    to int.
  @      // Note 2: BITS_PER_WORD = 64.
  @      (words[i / BITS_PER_WORD] >>> (int)(i % BITS_PER_WORD)) & 1);
  @ }
  @*/
```

Per word in the logical length of `words` (as defined by `wordsInUse`), the sequence isolates each of the 64 individual bits and stores them as element i of the sequence. This converts the array as seen in Figure 1 to the sequence as seen in Figure 2.
Unlike the logical length of a bitset, the length of our sequence is always a multiple of 64, as `BITS_PER_WORD` equals 64 and the length of the sequence equals `wordsInUse*BITS_PER_WORD`. However, as was discussed earlier, any bit outside the logical length of a bitset is set to 0, which is also the case in this logical representation.

4.4 The `set(int)` method

The `set(int bitIndex)` method sets the specified bit to 1, provided that `bitIndex` is non-negative. The value of all other bits remains unchanged. Listing 8 shows the method and its contract.

Listing 8: The contract for the `set(int)` method, as well as the method body.

```
/*@
  @ normal_behaviour
  @ requires bitIndex >= 0;
  @ ensures wordsToSeq()[bitIndex] == 1;
  @ ensures (\forall \bigint j; 0 <= j < \old(wordsToSeq()).length & j != bitIndex;
  @         wordsToSeq()[j] == \old(wordsToSeq())[j] );
  @ ensures \old(wordsToSeq()).length < wordsToSeq().length ==>
  @   (\forall \bigint k;
  @     \old(wordsToSeq()).length <= k < wordsToSeq().length & k != bitIndex;
  @        wordsToSeq()[k] == 0
  @   );
  @*/
public void set(int bitIndex) {
    if (bitIndex < 0)
        throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);

    int wordIndex = wordIndex(bitIndex);
    expandTo(wordIndex);

    words[wordIndex] |= (1L << bitIndex); // Restores invariants

    checkInvariants();
}
```

After the method terminates, the specified bit must equal 1. This is given by `wordsToSeq()[bitIndex] == 1;`. The other two `ensures` clauses specify what the *other* bits should be. First, all bits (except `bitIndex`) that were defined in the sequence *prior* to the method being called should still be defined, and these should equal the value *after* the method terminates. Similarly, if the new sequence is longer than the old sequence, then all these new bits that are not at position `bitIndex` should equal 0, as this is the default value for a bit that was not in the defined length of the (original) bitset.

The `set(int)` method calls two other large methods, which we have given contracts: `expandTo(int)` and `ensureCapacity(int)`. These are detailed below.

4.5 The private `expandTo(int)` method

The `expandTo(int wordIndex)` method is a helper method, and is called by methods that set bits to 1. The invariant is true when it is called. `expandTo(int wordIndex)` makes sure that `wordIndex` fits within both the logically defined length (by increasing `wordsInUse`) and the actual length of the `words` array (by increasing `words.length`). The method's contract and body are visible in Listing 9:

Listing 9: The contract for the `expandTo(int)` method, as well as the method body.

```
/*@
  @ normal_behaviour
  @ requires
  @     wordIndex >= 0 & wordIndex <= Integer.MAX_VALUE/BITS_PER_WORD; // BITS_PER_WORD = 64
  @ requires \invariant_for(this);
  @
  @ ensures wordIndex < \old(wordsInUse) ==>
  @     words == \old(words) & wordsInUse == \old(wordsInUse);
  @ ensures wordIndex >= \old(wordsInUse) ==> wordsInUse == wordIndex+1;
  @ ensures wordIndex < words.length; // Implies: wordsInUse <= words.length (invariant)
  @ // Parts required to restore the invariant:
  @ ensures (\forall \bigint i; 0 <= i < \old(wordsInUse); words[i] == \old(words[i]));
  @ ensures (\forall \bigint i; \old(wordsInUse) <= i < words.length; words[i] == 0);
  @ ensures words != null & words.length >= \old(words).length;
  @ ensures wordsInUse <= (Integer.MAX_VALUE/BITS_PER_WORD + 1);
  @ ensures words.length <= 2*(Integer.MAX_VALUE/BITS_PER_WORD + 1);
  @ ensures (\forall \bigint i; 0 <= i < words.length; \dl_inLong(words[i]) );
  @ assignable words, wordsInUse, sizeIsSticky;
  @ helper
  @*/
private void expandTo(int wordIndex) {
    int wordsRequired = wordIndex+1;
    if (wordsInUse < wordsRequired) {
        ensureCapacity(wordsRequired);
        wordsInUse = wordsRequired;
    }
}
```

The invariant is valid when `expandTo(int)` is called. `wordIndex` is the result of some integer `bitIndex` divided by 64, meaning that it is at most `Integer.MAX_VALUE/64`. After the method terminates, there are two different scenarios:

First, where `wordIndex` is smaller than `wordsInUse` before the method is called. In this case, `wordIndex` already fitted in the logically defined length of `words`, and therefore nothing is changed in `words` or `wordsInUse`. In this case, the invariant is still true.

Alternatively, the logically defined length of `words` has been increased. `wordsInUse` is increased, and `ensureCapacity(int)` may increase the length of `words.length` to fit `wordIndex`. By increasing `wordsInUse` without setting any new bits, `words[wordsInUse-1] != 0` is no longer true, and therefore the invariant is temporarily broken. The invariant is restored in `set(int)`, as we set a bit in `words[wordIndex]` (and `wordIndex = wordsInUse-1`). In order to prove that the invariant is restored in `set(int)`, we specifically add the clauses from the invariant here that are still true once `expandTo(int)` terminates, such as the bounds of `words.length` (see Listing 20).

4.6 The private `ensureCapacity(int)` method

The `ensureCapacity(wordsRequired)` method expands the `words` array if `wordsRequired` does not fit in the array. The method's contract and body are visible in Listing 10:

Listing 10: The contract for the `ensureCapacity(int)` method, as well as the method body.

```
/*@
  @   normal_behaviour
  @     requires wordsRequired >= 0 & wordsRequired <= (Integer.MAX_VALUE/BITS_PER_WORD + 1);
  @     ensures words.length >= wordsRequired;
  @     ensures wordsToSeq() == \old(wordsToSeq());
  @     ensures \old(words).length <= words.length;
  @     ensures (\forall \bigint i; 0 <= i < \old(words).length;
  @                 \old(words[i]) == words[i]);
  @     ensures \old(words.length) < words.length ==> (\forall \bigint i;
  @        \old(words.length) <= i < words.length; words[i] == 0);
  @     assignable words, sizeIsSticky;
  @*/
private void ensureCapacity(int wordsRequired) {
    if (words.length < wordsRequired) {
        // Allocate larger of doubled size or required size
        int request = Math.max(2 * words.length, wordsRequired);
        words = Arrays.copyOf(words, request);
        sizeIsSticky = false;
    }
}
```

Unlike the `expandTo(int)` method, this method *does* preserve the invariant. The method may make the `words` array longer, but does not change values: elements that already existed in `words` remain the same, and all new elements are set to 0, as is default in `BitSet`.
By using `assignable words, ...;` the method contract shows to `expandTo(int)` that `wordsInUse` is not changed, while the method contract shows that the values within the array are also not altered.

4.7 The `clear()` method

The `clear()` method sets every bit in the bitset to 0. It is a simple method, but allows us to demonstrate a loop invariant. The method and its contract are visible in Listing 11:

Listing 11: The contract for the `clear()` method, as well as the method body.

```
/*@
  @ normal_behaviour
  @ requires true;
  @ ensures (\forall \bigint i; 0 <= i < wordsToSeq().length; wordsToSeq()[i] == 0);
  @*/
public void clear() {
    /*@
      @ maintaining wordsInUse <= words.length;
      @ maintaining (\forall \bigint i; wordsInUse <= i < words.length; words[i] == 0);
      @ maintaining wordsInUse >= 0;
      @ decreasing wordsInUse;
      @ assignable words[*], wordsInUse;
      @*/
    while (wordsInUse > 0)
        words[--wordsInUse] = 0;
}
```

When the method terminates, `wordsInUse` equals 0. This means that the `ensures` clause is trivial:

`wordsToSeq().length` equals `wordsInUse*64`, which means that `0 <= i < wordsToSeq().length;` is equivalent to `0 <= i < 0`. Provided that we reach the normal point of termination of the method (and no exception is raised), the `ensures` clause will always be true.

Before the loop is started, the class invariant is true. This tells us that `0 <= wordsInUse <= words.length` and that all `words[i]` from `words[wordsInUse]` onwards equal 0. As the loop iterates, `wordsInUse` is lowered by 1 per iteration, and another element of `words` equals 0. We use `decreasing wordsInUse;` to show that the loop eventually halts.

Unlike the loop invariant, the method contract does not specify that all elements of `words` equals 0. This has two reasons.

First of all, to create the a layer of abstraction between implementation and specification, as discussed before. But secondly, this is implicit in the class invariant: all elements of the `wordsToSeq()` sequence, and thus by extension the bitset, equal 0. This means that `wordsInUse` must equal 0, else `(wordsInUse == 0 | words[wordsInUse - 1] != 0)` would not hold. The class invariant further tells us that all elements from `words[wordsInUse]` onwards must equal 0. This then tells us that indeed every element of `words` must equal 0.

Verification of this method is largely trivial: KeY can verify the correctness of this method with almost no human interactions, and unlike methods such as `set(int)`, KeY can do so without requiring extensions. (See Section 6.3.) As such, we do not discuss the verification of `clear()` in Section 6. A proof file for this verification is provided in [Tat23].

4.8 The `get(int,int)` method

The `get(int,int)` 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`.

As we will show in Section 5.1, the `get(int,int)` method has a bug in it in its current form. Assuming that this bug is resolved, `get(int,int)` is an interesting method to look at for formal verification. It is one of the larger methods in the `BitSet` class, and verification requires a loop invariant. Furthermore, the proof involves comparing two different sequences, namely the original sequence and the sequence of a new BitSet.

The contract for this method can be seen in Listing 12:

Listing 12: 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);
  @ assignable \nothing;
  @*/
public BitSet get(int fromIndex, int toIndex) { .. }
```

The invariant must hold for the resulting bitset once the method has terminated. This is specified with `ensures \invariant_for(\result);`.

The last two ensures clauses specify that the resulting bitset contains the bits it should. Firstly, every element in result.wordsToSeq() should correspond with the same element in the original this.wordsToSeq(). Next, if an element i is out of the scope of either sequence, then that element should equal 0 in the other sequence. As an example: Say 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 the original bitset should equal 0, as this the default value of a bit that is outside of the logical length of the BitSet.
Finally, the assignable \nothing; clause indicates that the current object is not changed in any way.

4.9 A comparison with a different formal specification of BitSet

As mentioned in the introduction, a formal specification for the BitSet class was previously made by Gary Leavens [Lea02]. We will briefly discuss some of the interesting similarities and differences between the old specification and our specification.
Leavens' specification was written between 1998 and 2002, which was before OpenJDK was released. As a result, the specification has been written without access to the implementation details of the BitSet class.
This specification represents the bitset using a mathematical set of integers called trueBits, a model variable[4]. The trueBits sets contains an integer i if and only if i is set to true in the bitset. In order to ensure that the set can only feature integers the normal bitset could store, the class invariant states that each integer i in trueBits must be greater than or equal 0 and smaller than some integer capacity, also a model variable.
The approach used with trueBits is fundamentally similar to our approach using wordsToSeq(). In both cases, the actual contents of the bitset are represented in some *logical* representation, where a bit can only be set in the logical representation if and only if it is also set in the actual bitset. Both approaches have set a limit to the size of the logical representation: the largest integer that could be contained within trueBits must be smaller than capacity, while wordsToSeq()'s length is wordsInUse*BITS_PER_WORD.
Interestingly, Leavens' specification does not appear to be correct for the version of the BitSet class discussed in this thesis.
First of all, the range of values that trueBits can store does not match the range that BitSet can store. The class invariant states that each integer i in trueBits must be smaller than the model integer capacity. However, capacity is a normal Java integer, and therefore must comply with regular Java bounds of Integer.MIN_VALUE and Integer.MAX_VALUE. As a result, the largest integer that can be stored in trueBits is Integer.MAX_VALUE-1, while BitSet can also store Integer.MAX_VALUE. In Section 5 we will discuss issues this causes, but in BitSet's current form setting the Integer.MAX_VALUE bit is still possible, and thus it should also be possible in the *logical* representation of BitSet.
Next, Leavens' specification of the get(int,int) method is not satisfied by the current implementation of the method. In Leaven's specification, the get(int, int) method returns a bitset where the requested bits are in the same position in the resulting bitset (\result) as they were in the

[4] A model variable is a variable that only exists in the JML specification, and is used in a similar way to how a model method is used.

original bitset. However, as we showed in the example in Section 3, this is not the case: When we call `bset.get(10,20)`, the value of the bit stored at index 10 in `bset` is now stored at position 0 in `\result`, not position 10 as specified by Leavens.

5 Issues in `BitSet`

Through the use of formal specification combined with testing, we discovered a number of issues that are currently present in the `BitSet` class. We discuss these issues here, and suggest potential solution directions.

5.1 A bug in `get(int,int)`

The first bug occurs in the `get(int fromIndex, int toIndex)` method[5]. The first set of lines from this method are visible in Listing 13:

Listing 13: Beginning of the `get(int, int)` method, where the first bug occurs.

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

  checkInvariants();

  int len = length();

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

  // An optimization
  if (toIndex > len)
    toIndex = len;
  ...
```

The `length()` method returns the position of the most significant bit set to 1 plus 1.[6] For example, if the user sets bit 200 in a previously empty BitSet, then the `length()` method will return 201. 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`. This is by itself not necessarily an issue, depending on your interpretation of the specification of `length()`, but does cause an issue in the `get(int,int)` method.

Listing 14: Example of how the bug can occur with `get(int,int)`.

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

Listing 14 shows an example of this bug occurring in `get(int,int)`. The expected behaviour would be that `result` is a bitset with with logical length 1000 and which has bit 999 set. Instead, `result` has logical length 0 and has no bits set.

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

[6] This is the intended behaviour. This is not true if wordsInUse is set incorrectly, see Section 5.2.

`bset.length()` returns `Integer.MIN_VALUE`, because `Integer.MAX_VALUE` is set. The check `len <= fromIndex` on line 9 in the `get(int,int)` method then always evaluates to true, as `Integer.MIN_VALUE` is smaller than any non-negative signed integer in Java. This means that `result` is the empty bitset. The actual behaviour of `bset.get(x,y)` is that the method always returns the empty bitset when `0 <= x <= y`, regardless of the value of `x` and `y`.

Interestingly, this bug does not appear to be an issue in other methods where `length()` is called. As an example, we show a section of `BitSet`'s public `previousSetBit(int)` method in Listing 15:

Listing 15: A part of the `previousSetBit(int)` method, `length()` is also called but this bug does not occur.

```
public int previousSetBit(int fromIndex) {
    ...
    int u = wordIndex(fromIndex);
    if (u >= wordsInUse)
        return length() - 1;
```

`length()` is only called when `u` is larger than or equal to `wordsInUse`. When `length()` overflows, the bit at position `Integer.MAX_VALUE` is set, in `words[Integer.MAX_VALUE/64]`. This means that `wordsInUse` equals `Integer.MAX_VALUE/64+1`, which is the upper bound we gave `wordsInUse` in Section 4.2. `fromIndex` is an integer, and thus is at most `Integer.MAX_VALUE`. As `wordIndex(fromIndex)` returns `fromIndex/64` here, `u` is at most `Integer.MAX_VALUE/64`. If the value of `length()` overflows, then `u` will always be smaller than `wordsInUse`, as `u` is always smaller than `Integer.MAX_VALUE/64+1`.
`get(int,int)` is the only method where `length()` is called without checking `wordsInUse`, and thus the only method where the `length()` may overflow when being called.

5.2 Bugs resulting from the `valueOf(..)` methods

The next issue stems from the `valueOf(..)` methods[7]. We focus on the `long[]` method, but the same bug exists for the overloaded methods for `LongBuffer`, `ByteBuffer` and `byte[]`. The code for this method is visible in Listing 16.

Listing 16: 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[] longs)` method takes in the `longs` array. Before calling the private constructor, `valueOf(long[])` lowers `n` until either `n` equals 0 or `longs[n-1]` contains at least 1 set bit. The first `n` elements of `longs` are then copied to and stored in the new bitset instance. The

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

value `n` is then used as `wordsInUse`. Because either `n = 0` or `longs[n-1] != 0`, the condition `(wordsInUse == 0 || words[wordsInUse - 1] != 0)` of the invariant made true. Similarly, this also ensures that the other two conditions of `checkInvariants()` are true. As a result, the call to `checkInvariants()` in the private constructor will always pass. The `valueOf(long[] longs)` method does not have any specific requirements for `longs`: Any non-null array `longs` will be converted to a BitSet.

While `checkInvariants()` passes, this method can create bitset instances that cause issues in other methods. Specifically, this occurs when the method passes a `longs` array to the private constructor that is larger than our defined bounds for `words.length` and `wordsInUse`. (See Section 4.2.) In Listing 17 we show an example of bitset created with such a `longs` array.

Listing 17: Example of how the bug can occur with `valueOf(long[])`.

```
1 static final int MAX_WIU = Integer.MAX_VALUE/64 + 1;
2 BitSet normal = new BitSet();
3 normal.set(0);
4 long[] largeArray = new long[2*MAX_WIU + 1];
5 largeArray[ largeArray.length - 1] = 1;
6 BitSet broken = BitSet.valueOf(largeArray);
7 broken.set(0); // to ensure that broken.get(0) equals normal.get(0).
```

The `MAX_WIU` is the bound of `wordsInUse` as defined in Section 4.2. The `BitSet` class can only access elements of the array up to `largeArray[MAX_WIU-1]`. As a result, the bit set in `largeArray[2*MAX_WIU]` on line 5 is *not* accessible to `broken`.

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

When we examine the resulting value from `length()` of `broken`, we find that the return value did not only overflow to `Integer.MIN_VALUE` (as we discussed previously), but has then gone back up to 1. This phenomenon is not limited to this example: an array with length `4*MAX_WIU+1` with the same bit set in the last word will also state that the logical length is 1, as in this case the resulting value of the `length()` has wrapped around *twice*. In fact, with this overflow it is possible to to have `length()` return *any* value in the bounds of a 32 bit signed integer. This overflow will happen whenever `wordsInUse` is higher than `MAX_WIU`, as `wordsInUse` is used to calculate the return value of `length()`. (See Listing 18)

Listing 18: The `length()` method. The return value is calculated 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 value of `BITS_PER_WORD * (wordsInUse-1)`, where `wordsInUse` is greater than `Integer.MAX_VALUE/64 + 1` (and `BITS_PER_WORD` equals 64), will always be larger than the maximum value that fits in an 32 bit signed integer.

This overflow issue in `length()` persists when interacting normally with the BitSet; if the user sets a bit $i > 0$ in `broken` 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,because the value of `wordsInUse` is higher than any value (`MAX_WIU` or lower) that `BitSet` would ever normally assign to it.

This issue in the `valueOf(..)` methods does not appear to be a mistake in the *implementation*. 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 `valueOf(LongBuffer)`, and then uses `BitSet`'s `toLongArray()` method to then convert it to a long array. The implementation of the methods also allows for this, provided that the last element of the long buffer has at least one bit set.

Instead, this issue is caused by a mistake in the (informal) *specification* of the methods. It also nicely demonstrates the usefulness of formal specifications: Having determined the (normal) bounds for `wordsInUse`, we were able to spot that this was a potential issue with the `valueOf(..)` methods, which we confirmed through testing, using these bounds.

5.3 Solution directions

Using the class invariant as discussed in Section 4.2, we can now discuss solution directions to the issues discussed previously. We split the discussion up into two main directions: One where the `BitSet` class still allows the user to set the `Integer.MAX_VALUE` bit, and one where that becomes forbidden. We will also discuss the advantages and disadvantages of both approaches.

Permit using the `Integer.MAX_VALUE` bit

In most cases, using the `Integer.MAX_VALUE` bit is fine. The main issues in the current implementation rise from `length()` and `get(int,int)`.

First of all, Java's documentation states that the `length()` method "[r]eturns the "logical size" of this BitSet: the index of the highest set bit in the BitSet plus one" [Bit]. In the case of the "highest set bit" being `Integer.MAX_VALUE`, this specification is at best ambiguous, as a negative value is not generally expected for a "logical size". The specification should clarify that either some special value is returned for this scenario (such as `Integer.MIN_VALUE`), or for example that the resulting value should be interpreted as an **un**signed integer.

Next, a two-line addition to the code can fix the bug in `get(int,int)`. We show this in Listing 19:

Listing 19: A possible solution to the bug in `get(int,int)`.

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

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

// An optimization
if (toIndex > len)
    toIndex = len;
...
```

Our fix is on the lines 3-4. This two-line change only corrects the internal implementation of the method, and does not affect the method specification or the class specification. As a reminder, the method goes up to *but not including* `toIndex`. As a result, the highest index that the method can ever access is `Integer.MAX_VALUE-1`, as `toIndex` can never be higher than `Integer.MAX_VALUE`. Because of this, there is no difference to the method between `length()` returning `Integer.MAX_VALUE` or returning `Integer.MAX_VALUE+1` (assuming this would not cause an overflow). In both cases, the comparison `toIndex > len` on line 11 will always evaluate to false.
Finally, in order to fix the issues caused by `valueOf(..)`, the class should prevent `wordsInUse` becoming too large. One way of doing this is by having `valueOf(..)` throw an `IllegalArgumentException` if the array is longer than `MAX_WIU`. Alternatively, the method either ignore or discard elements after `words[MAX_WIU-1]`. Both of these changes require a change in the methods' specification.

Forbid using the `Integer.MAX_VALUE` bit

Forbidding setting the `Integer.MAX_VALUE` bit immediately prevents the `length()` method overflowing in normal bitset instances. As a result, the bug in `get(int,int)` is then also immediately solved.
This change also solves an issue that exists between methods with one parameter, and those with two. We take `get(int)` and `get(int,int)` as our example. Using `get(int)`, we can access the bit at index `Integer.MAX_VALUE`. However, we cannot do the same in `get(int fromIndex, int toIndex)`, because the method does not access the `toIndex` bit. By forbidding access to the `Integer.MAX_VALUE` bit, both one parameter and two parameter methods can access the same bits within the bitset.
The issue caused by `valueOf(..)` is not automatically solved by prohibiting access to the `Integer.MAX_VALUE` bit. Instead, solutions can be used as discussed in the previous section. In this case, the implementation of `valueOf(..)` may need to take extra care when loading in a large array: if the `Integer.MAX_VALUE` bit is set in the array that is loaded in, then that would still cause `length()` to overflow.

Discussion

The first solution direction represents the smaller change to the `BitSet` class. The `get(int,int)` bug can be fixed internally without changing the method's specification. The `length()` method's specification will change, but only to clarify behaviour that already existed. Assuming it is specified that `Integer.MAX_VALUE` being set means `Integer.MIN_VALUE` is returned, it should not change the way `length()` is currently used.
On the contrary, banning setting the `Integer.MAX_VALUE` bit represents a big change in the `BitSet` class. It alters one of the most fundamental parts of the specification of the class, namely that "[t]he bits of a BitSet are indexed by non-negative integers." [Bit]. It requires a lot of methods to be changed both in specification and implementation, such as by having them raise an exception when the user tries to access the `Integer.MAX_VALUE` bit. This may also break existing code using the `BitSet` class that relies on using all 2^{31} bits. In our opinion, the main advantage of this alternative direction, aside from preventing the overflow in `length()`, is that two parameter methods such as `get(int,int)` can access the same set of bits as single parameter methods such as `get(int)` can. The changes to the `valueOf(..)` methods prevents bitset instances being created that do not behave as expected. The behaviour is not changed if the user calls the methods with a valid parameter, i.e.

such as an array that fits within the bounds of **MAX_WIU**. A user who uses a parameter that is too big for the class will now see a change, such as an exception being raised or part of the parameter being left out. In our view, this change is necessary, as the **broken** instance does not behave as expected by the class specification.

6 Towards Formal Verification

Full formal verification of `BitSet`'s correctness is not currently possible. First of all, due to the bug in `get(int,int)`, `BitSet` is currently not correct. More importantly, a major issue with formal verification is that any proofs obtained can be discarded if the code or the specification is changed. In the `BitSet` class, this is very likely to happen.[8] Not only does the `get(int,int)` method need fixing, but larger parts of the class may change depending on the chosen solution direction from the previous discussion.
That being said, our chosen theorem prover, KeY, also currently requires some improvements and extensions before it can be used to fully verify this class. We will discuss why these are needed, and then we will explain some of the rules we have come up with. We will use these rules to verify the correctness of the current implementation of the `set(int)` method, to demonstrate their usefulness. We also sketch out a proof for the `get(int,int)` method, by providing a loop invariant and explaining part of what is required to complete the proof of the method.

6.1 Background

We add the bounds that we found in Section 4.2 for `words.length` and `wordsInUse` to the class invariant. (See Listing 20.) For the rules we created, we need the information that each element of the `words` fits in the primitive type `long`. We were not able to show this by itself in KeY. Instead, we added this information to the invariant using the "escape-sequence"[ABB+16] `\dl_inLong(..)`. This is not an original part of JML, but is an extension that allows us to make short statements that KeY can understand.

Listing 20: The full class invariant, including our bounds. An extension of Listing 5.

```
/*@ 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) ) &&
  @  // wordsInUse is bounded by the last word BitSet can set a bit in:
  @  (wordsInUse <= (Integer.MAX_VALUE/BITS_PER_WORD + 1) ) && // +1 is to round up.
  @  // words.length is bounded by 2*wordsInUse's bound (See ensureCapacity.)
  @  (words.length <= 2*(Integer.MAX_VALUE/BITS_PER_WORD + 1) ) &&
  @  // For the various taclets we have added, we require the assumption that
  @  // each array element of words is inLong. However, we were not able to
  @  // automatically show this in KeY itself.
  @  (\forall \bigint i; 0 <= i < words.length; \dl_inLong(words[i]) );
  @*/
```

[8]We have reported the bugs to Oracle, and opened a pull request with our two-line fix for `get(int,int)`. See https://github.com/openjdk/jdk/pull/13388.

6.2 The KeY theorem prover

KeY is a theorem prover designed with Java formal verification in mind. It can take Java code annotated with JML as input, and converts it to Java Dynamic Logic (JavaDL) [BKW16]. Using this, we can then work to verify the correctness of the JML specifications.

When we load a method and contract into KeY, the first step is to run the 'Finish Symbolic execution' macro. This macro goes through the method in the same way Java would execute the method, but with generic parameters instead of specific values. The macro splits the proof goal up whenever multiple options exist, usually depending on the values of the parameters. As an example, when the macro encounters an if-statement in the code, it will split the proof into a goal where the condition in the if-statement was true and one where it was false. Array accesses are also split into three possible goals: one where the array is null (raising an `NullPointerException`), one where the array access is out of bounds (raising an `ArrayIndexOutOfBoundsException`), and one where the array access is valid and does not cause any issues.

Based on our selected settings (see Table 1), if another method is called (in the body of the current method) and this method has a contract, the prover uses this contract: it needs to show that the pre-conditions of the contract hold here, and then the post-condition of the contract is assumed to be true and can be used to continue the proof. Because we *assume* the other contract is true when we use it here, we should first verify that other contract before using it here.

In our selected settings, loop invariants are split off into two proof goals: The prover first needs to show that the invariant is true when we initially reach the loop. Next, we then need to prove that the loop holds after an abstract amount of iterations.

In Section 6.4, we discuss rules that we have added to the KeY ruleset. We show a simple example in Listing 21, to explain the format:

Listing 21: An example of a rule we created for our proofs.

```
// x | y = 0
// This is true iff x = 0 and y = 0.
orLongZero {
    \schemaVar \term int x;
    \schemaVar \term int y;
    \assumes(inLong(x), inLong(y) ==>)
    \find( moduloLong(binaryOr(x, y)) = 0)
    \sameUpdateLevel
    \replacewith( x = 0 & y = 0 )
    \heuristics ( userTaclets2 )
};
```

The `assumes` clause contains terms or formulas that must be present in the current proof in order to apply the goal. The user then clicks on the term or formula in the `find(..)` clause, and this rule will appear as an option, provided that the `assume` clause is present. The `replacewith(..)` clause then shows what the selected term or formula is replaced with. Alternatively, the rule could have an `add(..)` clause, which adds extra information to the current goal.

For our verification, we use KeY version 2.10.0. The settings used are visible in Table 1.

Table 1: The Proof Search Strategy (left) and Taclet Options (right) used in KeY.

Max. Rule Applications	Various values*	JavaCard	Off
Stop at	Default	Strings	On
One Step Simplification	Disabled	Assertions	Off*
Proof splitting	Delayed	Bigint	On
Loop treatment	Invariant (Loop scope)	Initialisation	disableStaticInitialisation
Block treatment	Internal contract	intRules	javaSemantics
Method treatment	Contract	integerSimplification.	Full
Merge point statements	Merge	javaLoopTreatment	Efficient
Dependency contracts	On	mergeGenerateIsWeak.	Off
Query treatment	Off	methodExpansion	modularOnly
Expand local queries	On	modelFields	treatAsAxiom
Arithmetic treatment	Basic / DefOps*	moreSeqRules	On
Quantifier treatment	No splits with progs	permissions	Off
Class axiom rule	Off	programRules	Java
Auto induction	Off	reach	On
User-specific taclet sets	All off	runtimeExceptions	Ban
		sequences	On
		wdChecks	Off
		wdOperator	L

We will elaborate on the options marked with an asterisk (*):

- Max. Rule Applications: Depending on the situation, we want to use different amounts of rule applications. If we want to have KeY work on a lot of different goals, but want to avoid KeY getting stuck on one goal it cannot prove automatically for too long, then we may lower the rule count. If we are working on one specific goal and know it can be proven automatically from here, then we may increase the count.

- Arithmetic treatment: When using the 'Finish Symbolic execution' macro, we want to use the Basic option. If the DefOps option is on, then the macro will try to simplify calculated values, which results in proof goals being a lot less human-readable.
  As an example: the bound `MAX_WIU`, which equals `Integer.MAX_VALUE/64 +1`, is simplified to `-2147483648 + (1 + (2147483648 + jdiv(2147483647, 64)) % 4294967296) % 4294967296`. This change makes little to no difference to KeY, but does make it a lot harder for us as users to determine what we are working with.

- Assertions: The code features assertions made in Java (`assert(..)`) in `checkInvariants()`. However, these assertions have all been used in our class invariant. Therefore, by verifying the correctness of the invariant, we prove that these assertions will also always pass.

6.3 Required extensions to KeY

In its current form, KeY does not support verification of code involving bitwise operations, such as in the `set(int)` or `get(int,int)` methods. Firstly, bit shift operations such as the << used in `set(int)`, cause KeY's 'Finish Symbolic execution' macro to get stuck in a loop, as it endlessly applies rules to the shift term. Workarounds do exist for this, such as manually unfolding the shift term or by hiding the term until the macro is done, but this comes at the cost of more manual interactions.

More importantly, KeY's ruleset is currently not complete for bitwise operators such as `binaryAnd` or `binaryOr`.[9] It has rules for simple cases such as `binaryOr(0, x)` or `binaryAnd(1, x)`, but not for two generic variables. As a result, it is not possible to verify the correctness of the `set(int)` and `get(int,int)` methods in the current form of KeY.

There are different options for solving this problem. The terms could be translated to an SMT solver (see Section 2), or we can add rules to KeY.

In our case, we chose for the second option, developing a set of narrow rules that allow us to verify the correctness of the `set(int)` method. It may be possible to develop a general theory involving `binaryAnd` and `binaryOr` operators, but in our case this does not appear to be necessary. As discussed earlier, we use our `wordsToSeq()` model method to represent the bitset as a sequence of individual bits. When talking about one element of the `wordsToSeq()`, we are discussing a single bit. We can use this knowledge to make less general, but more simple rules. We will discuss these rules below, as we discuss the verification of the `set(int)` method.

6.4 Verification of the `set(int)` method

As a reminder, the method body and contract of `set(int)` is listed in Listing 22, as well as the method contract of `expandTo(int)`:

Listing 22: The `set(int)` method, as well as the `expandTo(int)` method contract.

```
/*@ normal_behaviour
  @ requires
  @     wordIndex >= 0 & wordIndex <= Integer.MAX_VALUE/BITS_PER_WORD; // BITS_PER_WORD = 64
  @ requires \invariant_for(this);
  @
  @ ensures wordIndex < \old(wordsInUse) ==>
  @     words == \old(words) & wordsInUse == \old(wordsInUse);
  @ ensures wordIndex >= \old(wordsInUse) ==> wordsInUse == wordIndex+1;
  @ ensures wordIndex < words.length; // Implies: wordsInUse <= words.length (invariant)
  @ // Parts required to restore the invariant:
  @ ensures (\forall \bigint i; 0 <= i < \old(wordsInUse); words[i] == \old(words[i]));
  @ ensures (\forall \bigint i; \old(wordsInUse) <= i < words.length; words[i] == 0);
  @ ensures words != null & words.length >= \old(words).length;
  @ ensures wordsInUse <= (Integer.MAX_VALUE/BITS_PER_WORD + 1);
  @ ensures words.length <= 2*(Integer.MAX_VALUE/BITS_PER_WORD + 1);
  @ ensures (\forall \bigint i; 0 <= i < words.length; \dl_inLong(words[i]) );
  @ helper
  @*/
private void expandTo(int wordIndex) { .. }

..

/*@
  @   normal_behaviour
  @     requires
  @         bitIndex >= 0;
  @     ensures wordsToSeq()[bitIndex] == 1;
  @     ensures (\forall \bigint i; 0 <= i < \old(wordsToSeq()).length & i != bitIndex;
  @             wordsToSeq()[i] == \old(wordsToSeq())[i] );
```

[9] This was also discussed in [Pfe17].

```
  @     ensures \old(wordsToSeq()).length < wordsToSeq().length ==>
  @         (\forall \bigint k;
  @             \old(wordsToSeq()).length <= k < wordsToSeq().length & k != bitIndex;
  @              wordsToSeq()[k] == 0
  @         );
  @*/
public void set(int bitIndex) {
    if (bitIndex < 0)
        throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);

    int wordIndex = wordIndex(bitIndex);
    expandTo(wordIndex); // helper method -> may brake the invar.

    words[wordIndex] |= (1L << bitIndex); // Restores invariants

    checkInvariants();
}
```

The contract for **expandTo(int)** has been proven correct. The proof for this can be found at [Tat23].

Newly created rules

KeY allows users to verify newly created rules. This can easily be done for rules that simplify specific cases. As an example, see Listing 23:

Listing 23: The `PowTwoNeqZero` rule, which adds additional information to the list of assumptions.

```
PowTwoNeqZero {
    \schemaVar \term int i;
    \assumes( i >= 0 ==> )
    \find( pow(2, i % 64) )
    \sameUpdateLevel
    \add( pow(2, i%64) != 0 ==>)
    \heuristics( userTaclets1 )
};
```

Rather than proving that this is true each individual time that comes up, we prove that the rule is correct once and can then simply use this rule. In order to verify that this rule is correct, we make use of KeY's existing ruleset.[10]

However, it is not possible to verify the rules we introduce to handle the `binaryOr` and `binaryAnd` operators inside of KeY, because we are *extending* KeY's ruleset in order to reason with these operators. When making these rules, we have attempted to limit their scope as much as possible, while still allowing us to use them to verify `set(int)`. This involves using the `inLong(..)` or `inInt(..)` escape-sequences in the assumptions, as well as limiting our rules to setting a single bit (as is done by `set(int)`).

This avoids the scenario where the rules may be correct when used in the context of a Java programme, but are not correct when allowing any mathematical number (as is possible in KeY). As an example, we look at the `orLongZero` rule (see Listing 21). In Java, the OR of two variables of type `long` equalling zero must mean that both of these variables separately both equal 0. This rule is used to verify part of the invariant, specifically `wordsInUse = 0 | words[wordsInUse-1] != 0`. In Java, `2*Long.MAX_VALUE+2` will overflow to 0. As a result, `moduloLong(binaryOr(2*Long.MAX_VALUE+2, 0))` equals 0. However, it is false to state that this implies that the *mathematical* integer `2*Long.MAX_VALUE+2` equals 0. By assuming `inLong(x)` in `orLongZero`, we prevent this situation, as `inLong(2*Long.MAX_VALUE+2)` is false.

[10] We do not discuss such rules here. We have provided proof files for these rules in [Tat23].

The following three rules that we have created apply to the `words[wordIndex] |= (1L << bitIndex);` operation. Note that `words[wordIndex]` is a 64 bit signed number, written in 2's complement. When this operation occurs, there are three different possible execution paths of the programme. Our rule splits the current proof goal up into these three paths, and is visible in Listing 24:

Listing 24: The `binaryOrSingleBit` rule, which splits the current proof up into 3 possible goals.

```
binaryOrSingleBit {
    \schemaVar \term int x;
    \schemaVar \term int i;
    \assumes( inLong(x), inInt(i), i >= 0, i <= 63 ==>)
    \find( binaryOr( x, moduloLong(shiftleft(1, i)) ) )
    \sameUpdateLevel
    // bit is already set -> OR has no effect.
    "Bit already set": \replacewith(x) \add(unsignedshiftrightJlong(x, i)%2 = 1 ==>);
    // Set the non-sign bit -> add the new bit = 2^i. i must be smaller than 63 for this.
    "Bit not yet set": \replacewith( x + pow(2, i) )
            \add( unsignedshiftrightJlong(x, i)%2 = 0, inLong(x + pow(2, i)), (x + pow(2, i))
     <= long_MAX, i < 63 ==>);
    // Set the sign bit -> convert the number from positive to negative 2's comp.
    // This is also only the case iff i = 63, so we add this to the list of assumptions.
    // Note: It is given that x >= 0, as otherwise the sign bit would already be set.
    "Set the sign bit": \replacewith(long_MIN + x)
            \add(unsignedshiftrightJlong(x, i)%2 = 0, x >= 0, inLong(long_MIN + x), (long_MIN
     + x) < 0, i = 63 ==>)
    \heuristics ( userTaclets2 )
};
```

We will explain each case:

- Firstly, the bit may already be set. In this case, the value of this bit, and therefore `words[wordIndex]` as whole, will the same as before the assignment. When we shift `words[wordIndex]` to the right (including the sign bit) and only take the bit at position `bitIndex`, then we know that it is 1.

- Next, if we set a bit that is not the sign bit, then we add $2^{(\text{bitIndex}\%64)}$ to the original value of `words[wordIndex]`. This applies both if `words[wordIndex]` is positive and if it is negative. Note also that no overflow will happen to `words[wordIndex]` here: the bit was not previously set, so adding $2^{(\text{bitIndex}\%64)}$ will only flip the bit at position `bitIndex` %64 to 1, and thus will not trigger a larger cascade of bits being flipped within `words[wordIndex]`. Furthermore, because we specify that `bitIndex` %64 is smaller than 63, $2^{(\text{bitIndex}\%64)}$ also does not overflow.

- Finally, if we set the sign bit, then `words[wordIndex]` goes from a positive number to a negative number. A similar logic applies here as above. Note that 2^{63} when stored in a primitive `long` in Java will overflow to `Integer.MIN_VALUE`.
  We know that the sign bit was not set previously, which tells us that `words[wordIndex]` was non-negative before the `|= ..` operation.

We have specified our contract using `wordsToSeq()` and therefore our proof also involves `wordsToSeq()`. To access a bit `k` located within an element of `words`, we shift the element `words[k/64]` to the right (including the sign bit) by `k % 64`, and then apply `& 1` [11] to this to get the specific bit `k`.

[11] Note: `n & 1` is equivalent to `n % 2` for an `n` that fits in a long. KeY's `binaryAndOne` rule uses this equivalence.

Altogether this is `(words[k/64] >>> (k%64)) & 1`.
If the `|= (1L << bitIndex)` has been applied to this element `words[k/64]`, then we need additional rules to deal with newly added `+ pow(2, k)` or `long_MIN` from the `binaryOrSingleBit` rule.

First, we look at the second case, where a bit other than the sign bit is set. We split this into two new cases, see Listing 25:

Listing 25: The **handleUnSHRlong** rule, which splits the current proof up into 2 possible goals, when a non-sign bit has been set.

```
handleUnSHRlong {
    \schemaVar \term int x;
    \schemaVar \term int i;
    \schemaVar \term int j;

    \assumes(inLong(x), inLong(x + pow(2, i)), inInt(i), i >= 0, i < 63, inInt(j),
                j >= 0, j <= 63, unsignedshiftrightJlong(x, i)%2 = 0 ==>)
    \find( unsignedshiftrightJlong(x + pow(2, i), j) )
    \sameUpdateLevel
    // With the SHR and the AND later, we will 'forget' the set bit.
    "i != j": \add(unsignedshiftrightJlong(x + pow(2, i), j) % 2
                       = unsignedshiftrightJlong(x, j) % 2, i != j ==>);
    // We are looking at the set bit -> it will be set to 1.
    "i = j": \add(unsignedshiftrightJlong(x + pow(2, i), j) % 2 = 1, i = j ==>)
    \heuristics ( userTaclets2 )
};
```

If our bit **k** does not equal `bitIndex`, then we know that this bit has not been changed with the `|= ..` operation. Furthermore, shifting `pow(2, i)` by an amount that does not equal **i** will always result in either an even number (`i > j`) or 0 (`i < j`). In both cases, this number `% 2` will equal 0. Therefore, isolating this bit will result in the same value as if this assignment was never made.
If our bit **k** equals `bitIndex`, then we know that this bit has been set by the `|= ..`. As a result, we know that shifting by **k** to the right and isolating the last bit will result in 1.
An important item in the list of assumptions is `unsignedshiftrightJlong(x, i)%2 = 0`. This says that the bit that we are looking at (by shifting **i** positions) is not set in the **x**, in other words the bit was not set in the original value of `words[wordIndex]`. Having used the `binaryOrSingleBit`, this is a given. However, without this assumption, the **handleUnSHRlong** rule would not be correct. As an example, we take $x = 1$ and $i = 0$. Here, the bit at position **i** is already set in **x**. When adding `pow(2, i)`, we get $x + 2^i = 1 + 2^0 = 1 + 1 = 2$. In the case of `i = j (= 0)`, this would then say that `unsignedshiftrightJlong(2, 0) % 2 = 1`, which is not correct. By assuming that the bit was not set in **x**, we know that the bit **i** is set in `x + pow(2, i)` by the `pow(2, i)`.

Next, we look at the third case, where we have set the sign bit. We again split this into two new cases, see Listing 26:

Listing 26: The **handleSignSHRLong** rule, which splits the proof up into 2 possible goals when the sign bit has been set.

```
handleSignSHRLong {
    \schemaVar \term int x;
    \schemaVar \term int j;
    \assumes(inInt(j), j >= 0, j <= 63, inLong(x), x >= 0 ==>)
    \find( unsignedshiftrightJlong(long_MIN + x, j) )
```

```
    \sameUpdateLevel
    // We know that the sign bit has been set -> the bit = 0.
    "j = 63": \replacewith( 1 ) \add(j = 63 ==>);
    // The sign bit has been set, but we not isolating the sign bit -> no change
    // compared to the orignal value.
    "j < 63": \add( unsignedshiftrightJlong(long_MIN + x, j) % 2 = unsignedshiftrightJlong(x,
     j) % 2, j < 63 ==>)
    \heuristics ( userTaclets2 )
};
```

Unlike in the `handleUnSHRlong` rule, we do not explicitly assume `unsignedshiftrightJlong(x, i)%2 = 0` here. Instead, we assume `x >= 0`. By assuming this, we state that the sign bit was not set in `x` (else `x` would be negative), and therefore we implicitly assume that `unsignedshiftrightJlong(x, 63)%2 = 0`. It is also worth noting that `long_MIN + x` is smaller than 0. `x` is at most `long_MAX`, and `long_MIN + long_MAX = -1`.
If our bit `k` equals `bitIndex`, then they both refer to the sign bit. By (unsigned) shifting a 64 bit number by 63, the result equals the sign bit. As we have set the sign bit (and as `long_MIN + x` is a negative number), it therefore equals 1.
If our bit `k` is not equal to `bitIndex`, then again this bit has not been altered, and thus equals the value of the bit in the original `words[wordIndex]`. As with the `handleUnSHRlong` rule, shifting `long_MIN` by less than 63 will result in an even number, and therefore `long_MIN % 2` will again equal 0.

Rules application in the `set(int)` proof

The first goal of the `set(int)` method is that the bit specified by `bitIndex` is set, or formally: `ensures wordsToSeq()[bitIndex] == 1;`.
In the case of the bit already being set before `set(int)` was called, `wordsToSeq()[bitIndex]` already equalled 1, and this will stay the case.
In the case of a non-sign bit being set, then the `handleUnSHRlong` rule is used. The goal of `i != j` is automatically closed, as both `i` and `j` refer to `bitIndex`. The goal of `i = j` again results in `1 = 1`, and so is closed.
Finally, if the sign bit is set, the rule `handleSignSHRLong` is used, and so the bit we refer to in this element of `words`, in other words `bitIndex % 64`, equals 63. The goal `j = 63` produces `1 = 1`, which is closed automatically. Similarly, the goal `j < 63` is closed automatically, as we create a contradiction when we we assume that `bitIndex % 64` both equals and is smaller than 63.

The second goal is that all other bits that already existed in the `wordsToSeq()` sequence remain unchanged, as specified with `ensures (\forall \bigint j; 0 <= j < \old(wordsToSeq()).length & j != bitIndex; wordsToSeq()[j] == \old(wordsToSeq())[j] );`. Here, the `|= ..` has been applied to the `wordsToSeq()`, while the `\old(wordsToSeq())` refers to the sequence before the `set(int)` method was called. For this goal, we use `expandTo(int)`'s contract: when the method terminates, the new value of `words[i]` equals the original value of `words[i]` *before* `expandTo(int)` was called, for all $0 \leq i \leq$ `\old(wordsInUse)`. This means that the logically defined elements of `words`, aside from `words[wordIndex]`, have not been changed. For these elements, this goal is trivial to prove: the entire word has not been changed (and `\old(wordsInUse)` $\leq$ `wordsInUse`, per `expandTo(int)`'s contract), so by extension the bit has not been changed and thus `\old(wordsToSeq())[j]` equals `wordsToSeq()[j]`.

If bit `j` is located within `words[wordIndex]`, then we need to show that this specific bit `j` has not been altered.
In the first case of the bit already being set, the `binaryOr(words[wordIndex], ..)` is replaced with `words[wordIndex]`, as it has not been changed. In this case, the entire `wordsToSeq()` sequence is the same as `\old(wordsToSeq())`, and bit `j` specifically is unchanged, leading to the goal being closed.
In the case of a non-sign bit being set, we use the `handleUnSHRlong` rule. The `i != j` case says that our bit at position `j` has not been changed, which is why the `+ pow(2, i)` is removed. This then shows that `wordsToSeq()[j]` equals the original value. The `i = j` case is closed automatically, as we have specified in the initial goal that `j != bitIndex`.
In the case of the sign bit being set, we know that bit `j` in `words[wordIndex]` refers to a *non*-sign bit, so we know that `j % 64` is smaller than 63. This then automatically causes a contradiction with `j = 63` case from `handleSignSHRLong`, resulting in that case being closed automatically. In the case with `j < 63`, we again can remove the added part from the `binaryOr`, in this case removing the `long_MIN`, telling us that `wordsToSeq()[j]` remains unchanged.

The third goal is that all new bits in the `wordsToSeq()` sequence that equal 0, aside from the bit as position `bitIndex`. Formally this is: `ensures \old(wordsToSeq()).length < wordsToSeq().length ==> (\forall \bigint k; \old(wordsToSeq()).length <= k < wordsToSeq().length & k != bitIndex; wordsToSeq()[k] == 0);` As a reminder, bits that were not previously defined in a bitset are set to 0 by default.
As with the previous goal, we use `expandTo(int)`'s contract: After `expandTo(int)` terminates, `words[i]` equals 0 for all `\old(wordsInUse)` $\leq$ `i` $\leq$ `words.length`. As with the previous goal, if bit `k` is not located in `words[wordIndex]`, then the element of `words` it is in equals 0, and thus bit `k` is also set to 0.
If the bit `k` is located within `words[wordIndex]`, then we again need to show that bit `k` has not been set and thus equals 0. This proof is analogous to the proof as explained for the second goal, but in each case we now show that `wordsToSeq()[k]` equals 0.

6.5 Proof sketch of the `get(int,int)` method

We will now sketch out the proof of the correctness of the `get(int,int)` method. For the purposes of this exposition, we assume the bug in the method has been fixed using our suggested fix. We also assume that the `valueOf(..)` bug cannot occur, and therefore that we can use the bounds to `words` and `wordsInUse` as shown in Listing 20. The contract and body of the method is visible in Listing 27. We mainly focus on the parts of the proof *not* related to the bitwise operators, as these are the parts that KeY in its current form can already verify.
A number of smaller methods are called in the `get(int,int)` method. These methods do not change pre-existing objects, and we have given these methods contracts. With the exception of the `length()` method, these contracts have all been verified in KeY with minimal or no human interaction. The completed proofs for these methods can be found in [Tat23]. The `length()` method uses shift operations, and as discussed previously this makes verification in the current form of KeY more difficult.

Listing 27: The contract and body of the `get(int,int)` method, including our suggested fix and our loop invariant.

```
/*@ 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);
  @ assignable \nothing;
  @*/
public BitSet get(int fromIndex, int toIndex) {
    checkRange(fromIndex, toIndex);

    checkInvariants();

    int len = length();

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

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

    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
```

```
66        result.wordsInUse = targetWords;
67        result.recalculateWordsInUse();
68        result.checkInvariants();
69
70        return result;
71    }
```

Local variables

After checking the parameters in lines 14-29, the method initialises a number of local variables. First, the `result` bitset is created, with a `words` array explicitly large enough to fit every bit between `fromIndex` and `toIndex`. `result.wordsInUse` is set to 0 until the `get(int,int)` method has copied the bits. The integer `targetWords` is the number of words to copy into `result.words`, and has the exact same value as `result.words.length`. `sourceIndex` is used to index elements of `this.words`. It initially refers to the element of `this.words` that contains the `fromIndex` bit. Finally, the boolean `wordAligned` indicates if `result` is aligned to the current bitset or not. 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 on line 37 is an adjusted version of `wordsToSeq()`: as `result.wordsInUse` is set to 0, we cannot use `result.wordsToSeq()` to refer to bits that have been copied. Instead, we use the loop iterator `i` to keep track of the bits that have been copied.
In order to verify the clauses from line 42 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 `this.words` (`wordsInUse`):

$$targetWords = \frac{toIndex - fromIndex - 1}{64} + 1 \leq wordsInUse$$

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` (line 28). Hence, the largest value `targetWords` can have is `wordsInUse`, in the case of:[12]

$$\frac{toIndex - fromIndex - 1}{64} + 1 = \frac{wordsInUse * 64 - 0 - 1}{64} + 1 \leq wordsInUse$$

Next, we can verify that the array accesses `words[sourceIndex]` and `words[sourceIndex+1]` in the loop body do not exceed the logically defined length of `words`, and by extension also not the *actual* length of `words`. For this, we make use of `targetWords`'s proven bound.
Firstly, we must prove the following:

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

This can be rewritten to:[13]

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

[12] Rounded using Java rules.

[13] Note that both `sourceIndex` and `targetWords` are calculated using `wordIndex(x)`, which will return `x/64` for $x \geq 0$.

We can write fromIndex as a multiple of 64 plus some offset, or fromIndex $= 64 * i + j$, where $0 \leq i \leq$ wordsInUse and $0 \leq j < 64$. $\frac{fromIndex}{64}$ then equals i, while $\frac{toIndex-fromIndex-1}{64}$ equals $\frac{toIndex-j-1}{64} - i$.[12] Altogether, this results in:

$$\frac{fromIndex}{64} + \left(\frac{toIndex - fromIndex - 1}{64} + 1\right) - 1 = i + \frac{toIndex - j - 1}{64} - i - 1 = \frac{toIndex - j - 1}{64} - 1$$

Using the bound for targetWords, we show that $sourceIndex + targetWords - 1$ indeed must be smaller than wordsInUse:

$$\frac{toIndex - j - 1}{64} - 1 \leq \frac{toIndex - 0 - 1}{64} - 1 < \frac{toIndex - 0 - 1}{64} \leq wordsInUse$$

Finally, if `((toIndex-1) & BIT_INDEX_MASK) < (fromIndex & BIT_INDEX_MASK)` (line 58) holds[14], then the boolean `wordAligned` must be false (as `(fromIndex & BIT_INDEX_MASK)` must be larger than 0). The `get(int,int)` method then uses `sourceIndex+1` to access the `this.words` array. In this case, the bound is tighter, as `sourceIndex` + `targetWords` must now be smaller than `wordsInUse-1` (rather than just `wordsInUse`). The proof for this is similar to the previous inequality, and can be proven automatically in KeY. We have replaced `n & 63` with `n % 64` in our proof file. These are analogous for non-negative `n`, but as discussed previously KeY does not support binaryAnd operations.
These lemmas have been verified in separate proof files using KeY, which can be found at [Tat23].

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`. The `wordsInUse == 0 || words[wordsInUse - 1] != 0` and `wordsInUse == words.length || words[wordsInUse] == 0` assertions from the class invariant may not be true for `result` when the method starts, as the method's purpose is to establish the invariant. Specifically, `wordsInUse` may be too high, which is the case if `words[wordsInUse-1]` is zero. All other clauses from the class invariant hold when `recalculateWordsInUse()` is called. To restore the class invariant, the method lowers `wordsInUse` to the most significant element of `result.words` that is not zero, or to zero if there is none.
At this point, the symbolic execution should be complete. In order to complete the proof, further bitwise operator rules are needed. These are needed to verify the loop invariant and to verify the `ensures` clauses of the `get(int,int)` method contract.

7 Conclusions and Further Research

In this thesis, we have discussed OpenJDK's `BitSet` class and formulated its formal specification. In the process of analysing the class, we have discovered bugs caused by integer overflows, one due to an error in the implementation in the code of the `get(int,int)` method, and one due to an oversight in the specification of the various `valueOf(..)` methods. We proposed a number of

[14] BIT_INDEX_MASK is a constant integer equalling 63.

different solution directions for these issues. We then discussed KeY, and explained why KeY's ruleset requires extensions before it can be used to verify the `BitSet` class. We gave examples of such extensions, and used them to verify the current version of the class' `set(int)` method. Finally, we discussed initial steps in order to verify the `get(int,int)` method.
The first big open question coming from this research is the future correctness of the `BitSet` class. If there are no other issues with the class aside from the two identified in this thesis, then it should be possible to verify the class' correctness once the developers have fixed the bugs. If it is not possible, then there may be more bugs yet to be discovered in the class. If the class does not significantly change, then the proof for the `set(int)` method may still be valid after the bugs are corrected, and then it should become possible to also verify the correctness of the (improved) `get(int,int)` method using the provided specification and proof sketch.
This leads to the second point of future research, regarding KeY's support for bitwise operators. While we have developed some *specific* rules in this research, and further specific rules could be developed for the verification of other methods such as `get(int,int)`, a long-term solution would be to implement *generalised* rules for bitwise operations within KeY, or through the use of an SMT solver.

References

[ABB+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016.

[BDL16] Arthur Blot, Pierre-Évariste Dagand, and Julia Lawall. From Sets to Bits in Coq. In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, volume 9613 of *Lecture Notes in Computer Science*, pages 12–28. Springer, 2016.

[Bit] BitSet (Java Platform SE 8). Last accessed 12 July 2023. https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html.

[BKW16] Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. *Dynamic Logic for Java*, pages 49–106. Springer, 2016.

[CLSE05] Yoonsik Cheon, Gary Leavens, Murali Sitaraman, and Stephen Edwards. Model Variables: Cleanly Supporting Abstraction in Design by Contract: Research Articles. *Softw. Pract. Exper.*, 35(6):583–599, May 2005.

[Daw09] Jeremy Dawson. Isabelle Theories for Machine Words. *Electronic Notes in Theoretical Computer Science*, 250(1):55–70, 2009. Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems (AVoCS 2007).

[DGRdB+15] Stijn De Gouw, Jurriaan Rot, Frank S de Boer, Richard Bubel, and Reiner Hähnle. 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*, pages 273–289. Springer, 2015.

[HBdBdG20] Hans-Dieter A. Hiep, Jinting Bian, Frank S. de Boer, and Stijn de Gouw. *A Tutorial on Verifying LinkedList Using KeY*, pages 221–245. Springer, 2020.

[jav] Primitive Data Types (The Java™ Tutorials). Last accessed 5 May 2023. https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html.

[Kro09] Daniel Kroening. Software verification. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, chapter 16, pages 505–532. IOS Press, February 2009.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. 523:175–188, 1999.

[Lea02] Gary T. Leavens. BitSet (JML and MultiJava documentation), 2002. Last accessed 5 May 2023. https://www.cs.ucf.edu/~leavens/JML-release/javadocs/java/util/BitSet.html.

[Pfe17] Wolfram Pfeifer. Specifying and verifying real-world java code with KeY - case study java.math.BigInteger. Bachelor thesis, Karlsruhe Institute of Technology, May 2017.

[Tat23] Andy S. Tatman. Formal Specification and Analysis of OpenJDK's BitSet class: Proof files, 2023. https://doi.org/10.5281/zenodo.8172478.

A Annotated `BitSet` class

A.1 Internal fields of the class

Listing 28: The relevant member variables of the `BitSet` class.

```
/*
 * BitSets are packed into arrays of "words."  Currently a word is
 * a long, which consists of 64 bits, requiring 6 address bits.
 * The choice of word size is determined purely by performance concerns.
 */
private static final int ADDRESS_BITS_PER_WORD = 6;
private static final int BITS_PER_WORD = 1 << ADDRESS_BITS_PER_WORD;
private static final int BIT_INDEX_MASK = BITS_PER_WORD - 1;

/* Used to shift left or right for a partial word mask */
private static final long WORD_MASK = 0xffffffffffffffffL;

/**
 * The internal field corresponding to the serialField "bits".
 */
private long[] words;

/**
 * The number of words in the logical size of this BitSet.
 */
private transient int wordsInUse = 0;
```

```java

/**
 * Whether the size of "words" is user-specified.  If so, we assume
 * the user knows what he's doing and try harder to preserve it.
 */
private transient boolean sizeIsSticky = false;
```

A.2 Class invariant

Listing 29: Our full class invariant of the `BitSet` class.

```java
/*@ 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) ) &&
  @  // wordsInUse is bounded by the last word BitSet can set a bit in:
  @  (wordsInUse <= (Integer.MAX_VALUE/BITS_PER_WORD + 1) ) && // +1 is to round up.
  @  // words.length is bounded by 2*wordsInUse's bound (See ensureCapacity.)
  @  (words.length <= 2*(Integer.MAX_VALUE/BITS_PER_WORD + 1) ) &&
  @  // For the various taclets we have added, we require the assumption that
  @  // each array element of words is inLong. However, we were not able to
  @  // automatically show this in KeY itself.
  @  (\forall \bigint i; 0 <= i < words.length; \dl_inLong(words[i]) );
  @*/
```

A.3 Annotated methods

Unless stated otherwise, the correctness of all of these method contracts have been verified, with proof files for each provided at [Tat23].

A.3.1 wordIndex(int)

Listing 30: The annotated `wordIndex(int)` method.

```java
/**
 * Given a bit index, return word index containing it.
 */
/*@ normal_behaviour
  @ requires bitIndex >= -1;
  @ ensures \old(bitIndex) >= 0 ==> \result == (\old(bitIndex) / 64);
  @ ensures \old(bitIndex) == -1 ==> \result == -1;
  @*/
private static /*@ strictly_pure @*/ int wordIndex(int bitIndex) {
    return bitIndex >> ADDRESS_BITS_PER_WORD;
}
```

A.3.2 checkInvariants()

Listing 31: The annotated `checkInvariants()` method.

```java
/**
```

```
 * Every public method must preserve these invariants.
 */
/*@ normal_behaviour
  @ ensures true;
  @ assignable \strictly_nothing; @*/
private void checkInvariants() {
    assert(wordsInUse == 0 || words[wordsInUse - 1] != 0);
    assert(wordsInUse >= 0 && wordsInUse <= words.length);
    assert(wordsInUse == words.length || words[wordsInUse] == 0);
    // By induction: forall i: wordsInUse <= i <= a.length(): words[i] = 0.
}
```

A.3.3 recalculateWordsInUse()

Listing 32: The annotated `recalculateWordsInUse()` method.

```
/**
 * Sets the field wordsInUse to the logical size in words of the bit set.
 * WARNING:This method assumes that the number of words actually in use is
 * less than or equal to the current value of wordsInUse!
 */
/*@
  @ normal_behaviour
  @ requires words != null;
  @ requires wordsInUse >= 0 && wordsInUse <= words.length;
  @ requires wordsInUse < words.length ==>
  @      (\forall \bigint i; wordsInUse <= i < words.length; words[i] == 0);
  @ requires (wordsInUse <= (Integer.MAX_VALUE/BITS_PER_WORD + 1) );
  @ requires (words.length <= 2*(Integer.MAX_VALUE/BITS_PER_WORD +1) );
  @ requires (\forall \bigint i; 0 <= i < words.length; \dl_inLong(words[i]));
  @ ensures \invariant_for(this);
  @ ensures wordsInUse <= \old(wordsInUse);
  @ assignable this.wordsInUse;
  @ helper
  @*/
private void recalculateWordsInUse() {
    // Traverse the bitset until a used word is found
    int i;
    /*@
      @ maintaining (\forall \bigint j; i < j < words.length; words[j] == 0);
      @ maintaining i < wordsInUse && i >= -1;
      @ decreasing i+1; // +1: At the end of the loop (if !break), i=-1.
      @ assignable \strictly_nothing;
      @*/
    for (i = wordsInUse-1; i >= 0; i--)
        if (words[i] != 0)
            break;

    wordsInUse = i+1; // The new logical size
}
```

A.3.4 The public BitSet constructors

Listing 33: The public `BitSet` constructors, with both public and private constructors.

```
/**
 * Creates a new bit set. All bits are initially {@code false}.
 */
/*@ public normal_behaviour
  @ requires true;
  @ ensures wordsToSeq() == \seq_empty;
  @ assignable \nothing;
```

```
  @*/
/*@ private normal_behaviour
  @ requires true;
  @ ensures words.length == 1;
  @ ensures wordsToSeq() == \seq_empty;
  @ assignable \nothing;
  @*/
public BitSet() {
    initWords(BITS_PER_WORD);
    sizeIsSticky = false;
}

/**
 * Creates a bit set whose initial size is large enough to explicitly
 * represent bits with indices in the range {@code 0} through
 * {@code nbits-1}. All bits are initially {@code false}.
 *
 * @param  nbits the initial size of the bit set
 * @throws NegativeArraySizeException if the specified initial size
 *         is negative
 */
/*@ public normal_behaviour
  @ requires nbits >= 0;
  @ ensures wordsToSeq() == \seq_empty;
  @ assignable \nothing;
  @*/
/*@ private normal_behaviour
  @ requires nbits >= 0;
  @ ensures nbits == 0 ==> words.length == 0;
  @ ensures nbits > 0 ==> words.length == ((nbits-1) / 64) + 1;
  @ ensures wordsToSeq() == \seq_empty;
  @ assignable \nothing;
  @*/
public BitSet(int nbits) {
    // nbits can't be negative; size 0 is OK
    if (nbits < 0)
        throw new NegativeArraySizeException("nbits < 0: " + nbits);

    initWords(nbits);
    sizeIsSticky = true;
}

private void initWords(int nbits) {
    words = new long[wordIndex(nbits-1) + 1];
}
```

A.3.5 ensureCapacity(int)

Listing 34: The annotated **ensureCapacity(int)** method.

```
/**
 * Ensures that the BitSet can hold enough words.
 * @param wordsRequired the minimum acceptable number of words.
 */
/*@
  @   normal_behaviour
  @     requires
  @         wordsRequired >= 0 & wordsRequired <= (Integer.MAX_VALUE/BITS_PER_WORD + 1);
  @     ensures words.length >= wordsRequired;
  @     ensures wordsToSeq() == \old(wordsToSeq());
  @     ensures \old(words).length <= words.length;
  @     ensures (\forall \bigint i; 0 <= i < \old(words).length;
  @                 \old(words[i]) == words[i]);
  @     ensures \old(words.length) < words.length ==> (\forall \bigint i;
  @        \old(words.length) <= i < words.length; words[i] == 0);
```

```
  @     assignable words, sizeIsSticky;
  @*/
private void ensureCapacity(int wordsRequired) {
    if (words.length < wordsRequired) {
        // Allocate larger of doubled size or required size
        int request = Math.max(2 * words.length, wordsRequired);
        words = Arrays.copyOf(words, request);
        sizeIsSticky = false;
    }
}
```

A.3.6 expandTo(int)

Listing 35: The annotated expandTo(int) method.

```
/**
 * Ensures that the BitSet can accommodate a given wordIndex,
 * temporarily violating the invariants.  The caller must
 * restore the invariants before returning to the user,
 * possibly using recalculateWordsInUse().
 * @param wordIndex the index to be accommodated.
 */
/*@ normal_behaviour
  @ requires wordIndex >= 0 & wordIndex <= Integer.MAX_VALUE/BITS_PER_WORD;
  @ requires \invariant_for(this);
  @
  @ ensures wordIndex < \old(wordsInUse) ==>
  @     words == \old(words) & wordsInUse == \old(wordsInUse);
  @ ensures wordIndex >= \old(wordsInUse) ==> wordsInUse == wordIndex+1;
  @ ensures wordIndex < words.length; // Implies: wordsInUse <= words.length (invariant)
  @ // Parts required to restore the invariant:
  @ ensures (\forall \bigint i; 0 <= i < \old(wordsInUse); words[i] == \old(words[i]));
  @ ensures (\forall \bigint i; \old(wordsInUse) <= i < words.length; words[i] == 0);
  @ ensures words != null & words.length >= \old(words).length;
  @ ensures wordsInUse <= (Integer.MAX_VALUE/BITS_PER_WORD + 1);
  @ ensures words.length <= 2*(Integer.MAX_VALUE/BITS_PER_WORD + 1);
  @ ensures (\forall \bigint i; 0 <= i < words.length; \dl_inLong(words[i]) );
  @ helper
  @*/
private void expandTo(int wordIndex) {
    int wordsRequired = wordIndex+1;
    if (wordsInUse < wordsRequired) {
        ensureCapacity(wordsRequired);
        wordsInUse = wordsRequired;
    }
}
```

A.3.7 checkRange(int,int)

Listing 36: The annotated checkRange(int,int) method.

```
/**
 * Checks that fromIndex ... toIndex is a valid range of bit indices.
 */
/*@ normal_behaviour
  @ requires fromIndex >= 0 && toIndex >= 0 && fromIndex <= toIndex;
  @ ensures true;
  @ assignable \strictly_nothing;
  @*/
private static void checkRange(int fromIndex, int toIndex) {
    if (fromIndex < 0)
        throw new IndexOutOfBoundsException("fromIndex < 0: " + fromIndex);
```

```
    if (toIndex < 0)
        throw new IndexOutOfBoundsException("toIndex < 0: " + toIndex);
    if (fromIndex > toIndex)
        throw new IndexOutOfBoundsException("fromIndex: " + fromIndex +
                                            " > toIndex: " + toIndex);
}
```

A.3.8 set(int)

Listing 37: The annotated set(int) method.

```
/**
 * Sets the bit at the specified index to {@code true}.
 *
 * @param  bitIndex a bit index
 * @throws IndexOutOfBoundsException if the specified index is negative
 * @since  1.0
 */
/*@
  @   normal_behaviour
  @     requires
  @         bitIndex >= 0;
  @     ensures wordsToSeq()[bitIndex] == 1;
  @     ensures (\forall \bigint i; 0 <= i < \old(wordsToSeq()).length & i != bitIndex;
  @             wordsToSeq()[i] == \old(wordsToSeq())[i] );
  @     ensures \old(wordsToSeq()).length < wordsToSeq().length ==>
  @         (\forall \bigint k;
  @             \old(wordsToSeq()).length <= k < wordsToSeq().length & k != bitIndex;
  @              wordsToSeq()[k] == 0
  @         );
  @*/
public void set(int bitIndex) {
    if (bitIndex < 0)
        throw new IndexOutOfBoundsException("bitIndex < 0: " + bitIndex);

    int wordIndex = wordIndex(bitIndex);
    expandTo(wordIndex);

    words[wordIndex] |= (1L << bitIndex); // Restores invariants

    checkInvariants();
}
```

A.3.9 clear()

Listing 38: The annotated clear() method.

```
 /**
 * Sets all of the bits in this BitSet to {@code false}.
 *
 * @since 1.4
 */
/*@
  @ normal_behaviour
  @ requires true;
  @ ensures (\forall \bigint i; 0 <= i < wordsToSeq().length; wordsToSeq()[i] == 0);
  @*/
public void clear() {
    /*@
      @ maintaining wordsInUse <= words.length;
      @ maintaining (\forall \bigint i; wordsInUse <= i < words.length; words[i] == 0);
      @ maintaining wordsInUse >= 0;
```

```
        @ decreasing wordsInUse;
        @ assignable words[*], wordsInUse;
        @*/
    while (wordsInUse > 0)
        words[--wordsInUse] = 0;
}
```

A.3.10 get(int,int)

Listing 39: The annotated get(int,int) method. Note: This contract has not been verified.

```
/**
 * Returns a new {@code BitSet} composed of bits from this {@code BitSet}
 * from {@code fromIndex} (inclusive) to {@code toIndex} (exclusive).
 *
 * @param  fromIndex index of the first bit to include
 * @param  toIndex index after the last bit to include
 * @return a new {@code BitSet} from a range of this {@code BitSet}
 * @throws IndexOutOfBoundsException if {@code fromIndex} is negative,
 *         or {@code toIndex} is negative, or {@code fromIndex} is
 *         larger than {@code toIndex}
 * @since  1.4
 */
/*@ 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);
  @ assignable \nothing;
  @*/
public BitSet get(int fromIndex, int toIndex) {
    checkRange(fromIndex, toIndex);

    checkInvariants();

    int len = length();

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

    /* Our suggested bug fix: */
    if (len < 0)
        len = Integer.MAX_VALUE;

    // An optimization
    if (toIndex > len)
        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;
  }
```

A.3.11 length()

Listing 40: The annotated **length()** method. Note: This contract has not been verified.

```
/**
 * Returns the "logical size" of this {@code BitSet}: the index of
 * the highest set bit in the {@code BitSet} plus one. Returns zero
 * if the {@code BitSet} contains no set bits.
 *
 * @return the logical size of this {@code BitSet}
 * @since  1.2
 */
/*@ normal_behaviour
  @ requires true;
  @ ensures \result >= 0 || \result == Integer.MIN_VALUE;
  @ ensures \result == 0 ==>
  @     wordsToSeq().length == 0;
  @ ensures \result != 0 ==> wordsToSeq()[\result-1] == 1 &
  @         (\forall \bigint i; \result-1 < i < wordsToSeq().length; wordsToSeq()[i] == 0);
  // Result is in the last word of the LOGICAL size of words[]. (words[wIU-1] != 0)
  @ ensures \result != 0 ==> (\result-1 < wordsToSeq().length && \result-1 >= wordsToSeq().length - BITS_PER_WORD);
  @*/
public  /*@ strictly_pure @*/ int length() {
    if (wordsInUse == 0)
        return 0;

    return BITS_PER_WORD * (wordsInUse - 1) +
        (BITS_PER_WORD - Long.numberOfLeadingZeros(words[wordsInUse - 1]));
```

```
}
```

A.4 Our `wordsToSeq()` model method

Listing 41: Our `wordsToSeq()` model method.

```
// Our method for converting the actual representation to the logical representation.
/*@ 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  // >>> is not defined for bigint.
  @         );
  @ }
  @*/
```

A.5 The unannotated methods relevant to the `valueOf(long[])` discussion.

A.5.1 `valueOf(long[])`

Listing 42: The unannotated `valueOf(long[])` method and constructor it uses.

```
/**
 * Creates a bit set using words as the internal representation.
 * The last word (if there is one) must be non-zero.
 */
private BitSet(long[] words) {
    this.words = words;
    this.wordsInUse = words.length;
    checkInvariants();
}

/**
 * Returns a new bit set containing all the bits in the given long array.
 *
 * <p>More precisely,
 * <br>{@code BitSet.valueOf(longs).get(n) == ((longs[n/64] & (1L<<(n%64))) != 0)}
 * <br>for all {@code n < 64 * longs.length}.
 *
 * <p>This method is equivalent to
 * {@code BitSet.valueOf(LongBuffer.wrap(longs))}.
 *
 * @param longs a long array containing a little-endian representation
 *        of a sequence of bits to be used as the initial bits of the
 *        new bit set
 * @return a {@code BitSet} containing all the bits in the long array
 * @since 1.7
 */
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));
}
```

A.5.2 `toLongArray()`

Listing 43: The unannotated **toLongArray()** method.

```
/**
 * Returns a new long array containing all the bits in this bit set.
 *
 * <p>More precisely, if
 * <br>{@code long[] longs = s.toLongArray();}
 * <br>then {@code longs.length == (s.length()+63)/64} and
 * <br>{@code s.get(n) == ((longs[n/64] & (1L<<(n%64))) != 0)}
 * <br>for all {@code n < 64 * longs.length}.
 *
 * @return a long array containing a little-endian representation
 *         of all the bits in this bit set
 * @since 1.7
 */
public long[] toLongArray() {
    return Arrays.copyOf(words, wordsInUse);
}
```

B Rules added to KeY

The **andJLongDef**, **orJLongDef**, and **unsignedShiftRightJlongDef** rules have been directly adapted from KeY's **Def** rules for **Int**.
The various **Pow** rules have been proven correct, with proof files provided in [Tat23].
The other 4 rules, **orLongZero**, **binaryOrSingleBit**, **handleSignSHRLong**, and **handleUnSHRlong** have been discussed in Section 6.4.

B.1 andJLongDef

Listing 44: The **andJLongDef** rule.

```
// Same as andJIntDef, but with moduloLong.
andJLongDef {
  \schemaVar \term int left;
  \schemaVar \term int right;

  \find ( andJlong(left, right) )
  \replacewith ( moduloLong(binaryAnd(left, right)) )
  \heuristics ( userTaclets1 )
};
```

B.2 orJLongDef

Listing 45: The **orJLongDef** rule.

```
// Same as orJIntDef, but with moduloLong.
orJLongDef {
  \schemaVar \term int left;
  \schemaVar \term int right;

  \find ( orJlong(left, right) )
  \replacewith ( moduloLong(binaryOr(left, right)) )
  \heuristics ( userTaclets1 )
};
```

B.3 PowTwoNeqZero

Listing 46: The `PowTwoNeqZero` rule.

```
PowTwoNeqZero {
    \schemaVar \term int i;
    \assumes( i >= 0 ==> )
    \find( pow(2, i % 64) )
    \sameUpdateLevel
    \add( pow(2, i%64) != 0 ==>)
    \heuristics( userTaclets1 )
};
```

B.4 PowTwoGreZero

Listing 47: The `PowTwoGreZero` rule.

```
PowTwoGreZero {
    \schemaVar \term int i;
    \assumes( i >= 0, i%64 < 63 ==> )
    \find( pow(2, i % 64) )
    \sameUpdateLevel
    \add( pow(2, i % 64) >= 0 & inLong(pow(2, i % 64)) ==>)
    \heuristics( userTaclets1 )
};
```

B.5 ModPowTwoNeqZero

Listing 48: The `ModPowTwoNeqZero` rule.

```
ModPowTwoNeqZero {
    \schemaVar \term int i;
    \assumes( i >= 0 ==> )
    \find( moduloLong(pow(2, i % 64)) )
    \sameUpdateLevel
    \add( moduloLong(pow(2, i%64)) != 0 ==>)
    \heuristics( userTaclets1 )
};
```

B.6 ModPowTwoGreZero

Listing 49: The `ModPowTwoGreZero` rule.

```
ModPowTwoGreZero {
    \schemaVar \term int i;
    \assumes( i >= 0, i%64 < 63 ==> )
    \find( moduloLong(pow(2, i % 64)) )
    \sameUpdateLevel
    \add( moduloLong(pow(2, i % 64)) >= 0 & inLong(moduloLong(pow(2, i % 64))) ==>)
    \heuristics( userTaclets1 )
};
```

B.7 orLongZero

Listing 50: The `orLongZero` rule.

```
// x | y = 0
// This is true iff x = 0 and y = 0.
orLongZero {
    \schemaVar \term int x;
    \schemaVar \term int y;
    \assumes(inLong(x), inLong(y) ==>)
    \find( moduloLong(binaryOr(x, y)) = 0)
    \sameUpdateLevel
    \replacewith( x = 0 & y = 0 )
    \heuristics ( userTaclets2 )
};
```

B.8 binaryOrSingleBit

Listing 51: The `binaryOrSingleBit` rule.

```
// We set a single bit in x using binaryOr.
binaryOrSingleBit {
    \schemaVar \term int x;
    \schemaVar \term int i;
    \assumes( inLong(x), inInt(i), i >= 0 ==>)
    \find( binaryOr( x, moduloLong(shiftleft(1, i)) ) )
    \sameUpdateLevel
    // bit is already set -> OR has no effect.
    "Bit already set": \replacewith(x) \add(unsignedshiftrightJlong(x, i)%2 = 1 ==>);
    // Set the non-sign bit -> add the new bit = 2^i. i must be smaller than 63 for this.
    "Bit not yet set": \replacewith( x + pow(2, i) ) \add( unsignedshiftrightJlong(x, i)%2 =
    0, (x + pow(2, i)) <= long_MAX, i < 63 ==>);
    // Set the sign bit -> convert the number from positive to negative 2's comp.
    // This is also only the case iff i = 63, so we add this to the list of assumptions.
    // Note: It is given that x >= 0, as otherwise the sign bit would already be set.
    "Set the sign bit": \replacewith(long_MIN + x) \add(unsignedshiftrightJlong(x, i)%2 = 0,
    x >= 0 , inLong(long_MIN + x), i = 63 ==>)
    \heuristics ( userTaclets2 )
};
```

B.9 unsignedShiftRightJlongDef

Listing 52: The `unsignedShiftRightJlongDef` rule.

```
// UNSIGNED shift right long:

// The normal rule, adapted from unsignedShiftRightJintDef:
unsignedShiftRightJlongDef {
  \schemaVar \term int left;
  \schemaVar \term int right;

  \find ( unsignedshiftrightJlong(left, right) )
  \replacewith (
    \if (left >= 0)
        \then (shiftrightJlong(left, right))
        \else (addJlong(shiftrightJlong(left, right),
                        shiftleftJlong(2,
                                       63 - right % 64)))
  )
  \heuristics ( userTaclets1 )
};
```

B.10 handleSignSHRLong

Listing 53: The handleSignSHRLong rule.

```
handleSignSHRLong {
    \schemaVar \term int x;
    \schemaVar \term int j;
    \assumes(inInt(j), j >= 0, j <= 63, inLong(x), x >= 0 ==>)
    \find( unsignedshiftrightJlong(long_MIN + x, j) )
    \sameUpdateLevel
    // We know that the sign bit has been set -> the bit = 0.
    "j = 63": \replacewith( 1 ) \add(j = 63 ==>);
    // The sign bit has been set, but we not isolating the sign bit -> no change
    // compared to the orignal value.
    "j < 63": \add( unsignedshiftrightJlong(long_MIN + x, j) % 2 = unsignedshiftrightJlong(x,
     j) % 2, j < 63 ==>)
    \heuristics ( userTaclets2 )
};
```

B.11 handleUnSHRlong

Listing 54: The handleUnSHRlong rule.

```
handleUnSHRlong {
    \schemaVar \term int x;
    \schemaVar \term int i;
    \schemaVar \term int j;

    \assumes(inLong(x), inLong(x + pow(2, i)), inInt(i), i >= 0, i < 63, inInt(j),
                j >= 0, j <= 63, unsignedshiftrightJlong(x, i)%2 = 0 ==>)
    \find( unsignedshiftrightJlong(x + pow(2, i), j) )
    \sameUpdateLevel
    // With the SHR and the AND later, we will 'forget' the set bit.
    "i != j": \add(unsignedshiftrightJlong(x + pow(2, i), j) % 2
                    = unsignedshiftrightJlong(x, j) % 2, i != j ==>);
    // We are looking at the set bit -> it will be set to 1.
    "i = j": \add(unsignedshiftrightJlong(x + pow(2, i), j) % 2 = 1, i = j ==>)
    \heuristics ( userTaclets2 )
};
```