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
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 1
2 Related Work 3
3 The `BitSet` class 3
4 Formal Specification 6
  4.1 Introduction to Java Modelling Language ........................ 7
  4.2 Class invariant for the `BitSet` class ............................. 9
  4.3 The `wordsToSeq()` model method ................................ 10
  4.4 The `set(int)` method .............................................. 11
  4.5 The private `expandTo(int)` method ................................. 12
  4.6 The private `ensureCapacity(int)` method ........................... 13
  4.7 The `clear()` method ............................................. 13
  4.8 The `get(int,int)` method ......................................... 14
  4.9 A comparison with a different formal specification of `BitSet` ............................ 15
5 Issues in `BitSet` 16
  5.1 A bug in `get(int,int)` ............................................ 16
  5.2 Bugs resulting from the `valueOf(..)` methods ...................... 17
  5.3 Solution directions .............................................. 19
6 Towards Formal Verification 21
  6.1 Background .................................................... 21
  6.2 The KeY theorem prover ........................................... 22
  6.3 Required extensions to KeY ....................................... 23
  6.4 Verification of the `set(int)` method ............................... 24
  6.5 Proof sketch of the `get(int,int)` method .......................... 29
7 Conclusions and Further Research

References

A Annotated BitSet class
A.1 Internal fields of the class ........................................... 34
A.2 Class invariant .......................................................... 35
A.3 Annotated methods ...................................................... 35
  A.3.1 wordIndex(int) ...................................................... 35
  A.3.2 checkInvariants() .................................................. 35
  A.3.3 recalculateWordsInUse() ........................................ 36
  A.3.4 The public BitSet constructors .................................. 36
  A.3.5 ensureCapacity(int) ............................................... 37
  A.3.6 expandTo(int) ........................................................ 38
  A.3.7 checkRange(int, int) ............................................ 38
  A.3.8 set(int) ............................................................ 39
  A.3.9 clear() ............................................................. 39
  A.3.10 get(int, int) ...................................................... 40
  A.3.11 length() ........................................................... 41
A.4 Our wordsToSeq() model method .................................. 42
A.5 The unannotated methods relevant to the valueOf(long[]) discussion.... 42
  A.5.1 valueOf(long[]) ................................................... 42
  A.5.2 toLongArray() ..................................................... 42

B Rules added to KeY
B.1 andJLongDef .......................................................... 43
B.2 orJLongDef ............................................................ 43
B.3 PowTwoNeqZero ........................................................ 44
B.4 PowTwoGreZero ........................................................ 44
B.5 ModPowTwoNeqZero .................................................... 44
B.6 ModPowTwoGreZero ..................................................... 44
B.7 orLongZero ............................................................ 44
B.8 binaryOrSingleBit ..................................................... 45
B.9 unsignedShiftRightJlongDef ........................................ 45
B.10 handleSignSHRLong .................................................. 46
B.11 handleUnSHRlong .................................................... 46
# 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 \( \text{sum}(x, y) \) 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 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 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/319a3b994703aac84df7bcde272adfcb3c0bbbf0/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 [HBdBG20] 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].

```java
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;
}
```

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

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.

```java
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  BitSet secondBSet = bset.get(10, 20);
8  value = secondBSet.get(10); // value = false.
9  value = secondBSet.get(0); // value = true.
10 len = secondBSet.length(); // len = 1.
11 bset.clear();
12 value = bset.get(10); // value = false.
13 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 \geq 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. 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.” 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). 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:

```
/*@
  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.

---

2https://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(\ldots) clause. As an example: for an integer x, \texttt{ensures x == \old(x);} says that the value of x should not have changed during the execution of that method.

The \texttt{normal\_behaviour} term indicates that, given the pre-condition, this method will always terminate normally and will not raise an exception.

The \texttt{assignable} clause indicates what parts of the heap can be changed. In this case, \texttt{assignable \strictly\_nothing;} tells us that nothing can be created or changed on the heap. One alternative is \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{ensures \invariant\_for\(\texttt{this}\);} 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.\footnote{Note: This method does not come from the \texttt{BitSet} class. This is an example we have created for this explanation.}

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>/*@ public normal_behaviour</td>
</tr>
<tr>
<td>2</td>
<td>@ requires a != null &amp; a.length &gt; 0;</td>
</tr>
<tr>
<td>3</td>
<td>@ ensures (result == false) =&gt; (\forall \texttt{bigint} i; 0 &lt;= i &lt; a.length; a[i] != x);</td>
</tr>
<tr>
<td>4</td>
<td>@ ensures (result == true) =&gt; !(\forall \texttt{bigint} i; 0 &lt;= i &lt; a.length; a[i] != x);</td>
</tr>
<tr>
<td>5</td>
<td>*/</td>
</tr>
<tr>
<td>6</td>
<td>public boolean find(\texttt{int[]} a, \texttt{int x}) {</td>
</tr>
</tbody>
</table>

Listing 4: An example of a simple method that uses a loop invariant.
```java
int i = 0;
/*@
 * maintaining i >= 0 & i <= a.length;
 * maintaining (\forall \text{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 \( \geq 0 \). 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 \( \text{words} \) 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 \( \text{checkInvariants()} \) method. These are the following:

- Firstly, either \( \text{wordsInUse} \) is 0, or the last word in the logically defined length of \( \text{words} \), i.e. \( \text{words}[\text{wordsInUse}-1] \), has at least one bit that is set to 1.

- Secondly, \( \text{wordsInUse} \) is in the range of \([0, \text{words.length}]\), inclusive.

- Finally, either \( \text{wordsInUse} \) equals the length of \( \text{words} \), or the first word outside the logical length of the \( \text{words} \) array, i.e. \( \text{words}[\text{wordsInUse}] \), 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 \( \text{BitSet} \)'s public methods. However, we have expanded the class invariant, as it can include more conditions: First of all, the \( \text{words} \) array is allocated in every constructor, and therefore is never null. Next, the third assertion from \( \text{checkInvariants()} \) appears to suggest that every word after \( \text{words}[\text{wordsInUse}-1] \) should equal 0. This is backed up by the implementation of other methods. As an example, we look at the \( \text{recalculateWordsInUse()} \) method. This helper method restores the invariant, by lowering \( \text{wordsInUse} \) until \( (\text{wordsInUse} == 0 || \text{words}[\text{wordsInUse} - 1] != \emptyset) \) is true. Here, the method’s specification assumes that every element above \( \text{words}[\text{wordsInUse} - 1] \) 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.

```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 \text{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.

```java
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 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.

```java
/*@
private model strictly_pure \seq wordsToSeq() {
  \return (\seq_def \bigint i;
    \bigint (\bigint)BITS_PER_WORD;
    // Note 1: Shifting is undefined for \bigint, hence why we cast (i % BITS_PER_WORD)
to int.
    (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.

```
/*@
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 
);
@*/
```

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

```java
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 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.

```java
/*@
 normal_behaviour
 requires wordsRequired >= 0 & wordsRequired <= (Integer.MAX_VALUE/BITS_PER_WORD + 1);
 ensures words.length >= wordsRequired;
 ensures \old(words).length <= words.length;
 ensures (\forall \bigint i; 0 <= i < \old(words).length; \old(words[i]) == \old(wordsToSeq())[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.

```java
/*@@
 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 < word.

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 to 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.

```java
/*@ normal_behaviour
  @ requires fromIndex >= 0 && fromIndex <= toIndex;
  @ ensures \result != this && \invariant_for(\result);
  @ ensures (\forall bigint i; 0 <= i < \result.wordsToSeq().length;
          (fromIndex + i < \result.wordsToSeq().length) \Rightarrow 
          \result.wordsToSeq()[fromIndex + i] == 0);
  @ ensures (\forall bigint i; \result.wordsToSeq().length <= i < (toIndex-fromIndex);
          (fromIndex + i < \result.wordsToSeq().length) \Rightarrow 
          \result.wordsToSeq()[fromIndex + i] == 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 $\text{get}(0, 100)$, 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 ($\text{result}$) 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 \texttt{bset.get(10,20)}, the value of the bit stored at index 10 in \texttt{bset} is now stored at position 0 in \texttt{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 \texttt{BitSet} class. We discuss these issues here, and suggest potential solution directions.

5.1 A bug in \texttt{get(int,int)}

The first bug occurs in the \texttt{get(int fromIndex, int toIndex)} method\footnote{This bug report has been accepted by Oracle, see JDK-8305734.}. The first set of lines from this method are visible in Listing 13:

Listing 13: Beginning of the \texttt{get(int, int)} method, where the first bug occurs.

```java
1 public BitSet get(int fromIndex, int toIndex) {
2     checkRange(fromIndex, toIndex);
3     checkInvariants();
4     int len = length();
5     // If no set bits in range return empty bitset
6     if (len <= fromIndex || fromIndex == toIndex)
7         return new BitSet(0);
8     // An optimization
9     if (toIndex > len)
10        toIndex = len;
11 ...```

The \texttt{length()} method returns the position of the most significant bit set to 1 plus 1.\footnote{This is the intended behaviour. This is not true if \texttt{wordsInUse} is set incorrectly, see Section 5.2.} For example, if the user sets bit 200 in a previously empty BitSet, then the \texttt{length()} method will return 201. If the user sets the bit at index \texttt{Integer.MAX_VALUE}, then the \texttt{length()} method will return the integer \texttt{Integer.MAX_VALUE+1}, which overflows to \texttt{Integer.MIN_VALUE}. This is by itself not necessarily an issue, depending on your interpretation of the specification of \texttt{length()}, but does cause an issue in the \texttt{get(int,int)} method.

Listing 14: Example of how the bug can occur with \texttt{get(int,int)}.

```java
1 BitSet bset = new BitSet(0);
2 bset.set(Integer.MAX_VALUE);
3 bset.set(999);
4 BitSet result = bset.get(0,1000);
```

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

```java
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. 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.

```java
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

---

7This bug report has been accepted by Oracle, see JDK-8311905.
value \( n \) is then used as \( \text{wordsInUse} \). Because either \( n = 0 \) or \( \text{longs}[n-1] \neq 0 \), the condition \( \text{wordsInUse} == 0 || \text{words}[\text{wordsInUse} - 1] \neq 0 \) of the invariant made true. Similarly, this also ensures that the other two conditions of \( \text{checkInvariants()} \) are true. As a result, the call to \( \text{checkInvariants()} \) in the private constructor will always pass. The \( \text{valueOf(long[] longs)} \) method does not have any specific requirements for \( \text{longs} \): Any non-null array \( \text{longs} \) will be converted to a BitSet.

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

Listing 17: Example of how the bug can occur with \( \text{valueOf(long[])} \).

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

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

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

Listing 18: The \( \text{length()} \) method. The return value is calculated using \( \text{wordsInUse} \).

```java
1 public /*@ strictly_pure @*/ int length() {
2     if (wordsInUse == 0)
3         return 0;
4
5     return BITS_PER_WORD * (wordsInUse - 1) + (BITS_PER_WORD - Long.numberOfLeadingZeros(words[wordsInUse - 1]));
6 }
```

The value of \( \text{BITS_PER_WORD} * (\text{wordsInUse}-1) \), where \( \text{wordsInUse} \) is greater than \( \text{Integer.MAX_VALUE/64 + 1} \) (and \( \text{BITS_PER_WORD} \) equals 64), will always be larger than the maximum value that fits in an 32 bit signed integer.

This overflow issue in \( \text{length()} \) 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 unsigned integer.

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

```
1  ...
2  int len = length();
3  if (len < 0)
4      len = Integer.MAX_VALUE;
5  
6  // If no set bits in range return empty bitset
7  if (len <= fromIndex || fromIndex == toIndex)
8      return new BitSet(0);
9  
10  // An optimization
11  if (toIndex > len)
12      toIndex = len;
13  ...  
```

Listing 19: A possible solution to the bug in `get(int,int)`. We show this in Listing 19:
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 normalbitset 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.\(^6\) 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 */
0  @ words != null &
1  @ // The first three are from checkInvariants:
2  @ (wordsInUse == 0 || words[wordsInUse - 1] != 0) &&
3  @ (wordsInUse >= 0 && wordsInUse <= words.length) &&
4  @ (wordsInUse == words.length || words[wordsInUse] == 0) &&
5  @ // Our addition to the invariant:
6  @ (wordsInUse < words.length ==>
7  @  (\forall \bigint i; wordsInUse <= i < words.length; words[i] == 0) ) &&
8  @ // wordsInUse is bounded by the last word BitSet can set a bit in:
9  @ (wordsInUse <= (Integer.MAX_VALUE/BITS_PER_WORD + 1) ) && // +1 is to round up.
10 @ // words.length is bounded by 2*wordsInUse's bound (See ensureCapacity.)
11 @ (words.length <= 2*(Integer.MAX_VALUE/BITS_PER_WORD + 1) ) &&
12 @ // For the various taclets we have added, we require the assumption that
13 @ // each array element of words is inLong. However, we were not able to
14 @ // automatically show this in KeY itself.
15 @ (\forall \bigint i; \emptyset <= i < words.length; \texttt{dl\_inLong(words[i])} );
16 @*/
```

\(^6\)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.

```java
1 // x | y = 0
2 // This is true iff x = 0 and y = 0.
3 orLongZero {  
4 \schemaVar \term int x;
5 \schemaVar \term int y;
6 \assumes(inLong(x), inLong(y) =>)
7 \find( moduloLong(binaryOr(x, y)) = 0)
8 \sameUpdateLevel
9 \replacewith( x = 0 & y = 0 )
10 \heuristics ( userTaclets2 )
11 );
```

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.

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

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 \( \text{MAX}_\text{WIU} \), which equals \( \text{Integer.MAX_VALUE}/64 + 1 \), is simplified to \(-2147483648 + (1 + (2147483648 + \text{jdiv}(2147483647, 64)) \% 4294967296) \% 4294967296\). 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 (\text{assert}(...) in \text{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 \text{set(int)} or \text{get(int, int)} methods. Firstly, bit shift operations such as the \text{<}< 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. 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.

```java
/*@ normal_behaviour */
private void expandTo(int wordIndex) { .. }
```

This was also discussed in [Pfe17].
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.\footnote{We do not discuss such rules here. We have provided proof files for these rules in [Tat23].}

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 = \emptyset | words[wordsInUse-1] != \emptyset`. In Java, `2*Long.MAX_VALUE+2` will overflow to 0. As a result, `modulo(Long(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.
The following three rules that we have created apply to the \texttt{words[wordIndex] \|= (1L \ll bitIndex);} operation. Note that \texttt{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 \texttt{binaryOrSingleBit} rule, which splits the current proof up into 3 possible goals.

```java
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( unsignedshiftrightLong(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( unsignedshiftrightLong(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.
  "Set the sign bit": \replacewith( long_MIN + x)
  \add( unsignedshiftrightLong(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 \texttt{words[wordIndex]} as whole, will the same as before the assignment. When we shift \texttt{words[wordIndex]} to the right (including the sign bit) and only take the bit at position \texttt{bitIndex}, then we know that it is 1.

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

- Finally, if we set the sign bit, then \texttt{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 \texttt{long} in Java will overflow to \texttt{Integer.MIN_VALUE}.

We know that the sign bit was not set previously, which tells us that \texttt{words[wordIndex]} was non-negative before the \|= \ldots \texttt{operation}.

We have specified our contract using \texttt{wordsToSeq()} and therefore our proof also involves \texttt{wordsToSeq()}. To access a bit \(k\) located within an element of \texttt{words}, we shift the element \texttt{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 \(k\).

\footnote{Note: \texttt{n \& 1} is equivalent to \texttt{n \% 2} for an \texttt{n} that fits in a long. KeY’s \texttt{binaryAndOne} rule uses this equivalence.}
Altogether this is \( \text{words}[k/64] \gg (k\%64) \& 1 \). If the \(|= (1L << \text{bitIndex})\) has been applied to this element \text{words}[k/64], then we need additional rules to deal with newly added \( + \text{pow}(2, \ k) \) or \text{long_MIN} from the \text{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 \text{handleUnSHRlong} rule, which splits the current proof up into 2 possible goals, when a non-sign bit has been set.

```plaintext
1 handleUnSHRlong {
  \schemaVar \term int x;
  \schemaVar \term int i;
  \schemaVar \term int j;
  \assumes (inLong(x), inLong(x + \text{pow}(2, i)), inInt(i), i >= 0, i < 63, inInt(j),
            j >= 0, j <= 63, unsignedshiftrightJlong(x, i)%2 = 0 =>)
  \find ( unsignedshiftrightJlong(x + \text{pow}(2, i), j) )
  \sameUpdateLevel
  // With the SHR and the AND later, we will 'forget' the set bit.
  "i != j": \add (unsignedshiftrightJlong(x + \text{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 + \text{pow}(2, i), j) % 2 = 1, i = j =>)
  \heuristics ( userTaclets2 )
};
```

If our bit \( k \) does not equal \text{bitIndex}, then we know that this bit has not been changed with the \(|= ..\) operation. Furthermore, shifting \( \text{pow}(2, \ i) \) 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 \text{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 \text{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 \text{words[wordIndex]}. Having used the \text{binaryOrSingleBit}, this is a given. However, without this assumption, the \text{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 \( \text{pow}(2, \ i) \), we get \( x + 2^i = 1 + 2^0 = 1 + 1 = 2 \). In the case of \( i = j (= 0) \), this would then say that \text{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 \( x + \text{pow}(2, \ i) \) by the \( \text{pow}(2, \ i) \).

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 \text{handleSignSHRLong} rule, which splits the proof up into 2 possible goals when the sign bit has been set.

```plaintext
1 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) )
};
```
We know that the sign bit has been set, so the bit = 0.

The sign bit has been set, but we not isolating the sign bit, so no change.

Replace with (1) add(j = 63 == >)

The 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 [i] 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 <= i <= \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) <= \old(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 \( \text{words}[\text{wordIndex}] \), 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 \( \text{binaryOr}(\text{words}[\text{wordIndex}], \ldots) \) is replaced with \( \text{words}[\text{wordIndex}] \), as it has not been changed. In this case, the entire \( \text{wordsToSeq()} \) sequence is the same as \( \text{old}(\text{wordsToSeq}()) \), 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 \( \text{handleUnSHRlong} \) rule. The \( i \neq j \) case says that our bit at position \( j \) has not been changed, which is why the \( + \text{pow}(2, i) \) is removed. This then shows that \( \text{wordsToSeq}()[j] \) equals the original value. The \( i = j \) case is closed automatically, as we have specified in the initial goal that \( j \neq \text{bitIndex} \).

In the case of the sign bit being set, we know that bit \( j \) in \( \text{words}[\text{wordIndex}] \) 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 \( \text{handleSignSHRLong} \), resulting in that case being closed automatically. In the case with \( j < 63 \), we again can remove the added part from the \( \text{binaryOr} \), in this case removing the \( \text{long}_\text{MIN} \), telling us that \( \text{wordsToSeq}()[j] \) remains unchanged.

The third goal is that all new bits in the \( \text{wordsToSeq}() \) sequence that equal 0, aside from the bit as position \( \text{bitIndex} \). Formally this is: \( \text{ensures } \text{old}(\text{wordsToSeq}()).\text{length} < \text{wordsToSeq}().\text{length} \Rightarrow (\forall \text{bigint } k; \text{old}(\text{wordsToSeq}()).\text{length} \leq k < \text{wordsToSeq}().\text{length} \& k \neq \text{bitIndex}; \text{wordsToSeq}()[k] = 0) \); 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 \( \text{expandTo(int)} \)’s contract: After \( \text{expandTo(int)} \) terminates, \( \text{words}[i] \) equals 0 for all \( \text{old}(\text{wordsInUse}) \leq i \leq \text{words}.\text{length} \). As with the previous goal, if bit \( k \) is not located in \( \text{words}[\text{wordIndex}] \), then the element of \( \text{words} \) it is in equals 0, and thus bit \( k \) is also set to 0.

If the bit \( k \) is located within \( \text{words}[\text{wordIndex}] \), 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 \( \text{wordsToSeq}()[k] \) equals 0.

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

We will now sketch out the proof of the correctness of the \( \text{get(int,int)} \) 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 \( \text{valueOf}() \) bug cannot occur, and therefore that we can use the bounds to \( \text{words} \) and \( \text{wordsInUse} \) 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 \( \text{get(int,int)} \) method. These methods do not change pre-existing objects, and we have given these methods contracts. With the exception of the \( \text{length()} \) 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 \( \text{length()} \) 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 \( \text{get(int,int)} \) method, including our suggested fix and
our loop invariant.

```java
/*@ normal_behaviour
@ requires fromIndex >= 0 && fromIndex <= toIndex;
@ 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) == \result.wordsToSeq()[i];
@ assignable \nothing;
@*/
public BitSet get(int fromIndex, int toIndex) {
  checkRange(fromIndex, toIndex);
  checkInvariants();
  int len = length();
  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* 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;
}

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:

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

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:\[\]

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

12 Rounded using Java rules.
13 Note that both sourceIndex and targetWords are calculated using wordIndex(x), which will return x/64 for x ≥ 0.
We can write fromIndex as a multiple of 64 plus some offset, or \( \text{fromIndex} = 64 \times i + j \), where \( 0 \leq i \leq \text{wordsInUse} \) and \( 0 \leq j < 64 \). then equals \( i \), while \( \frac{\text{toIndex} - \text{fromIndex} - 1}{64} \) equals \( \frac{\text{toIndex} - \text{fromIndex} - 1}{64} - i \). Altogether, this results in:

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

Using the bound for targetWords, we show that \( \text{sourceIndex} + \text{targetWords} - 1 \) indeed must be smaller than \( \text{wordsInUse} \):

\[
\frac{\text{toIndex} - j - 1}{64} - 1 \leq \frac{\text{toIndex} - 0 - 1}{64} - 1 < \frac{\text{toIndex} - 0 - 1}{64} \leq \text{wordsInUse}
\]

Finally, if \(((\text{toIndex}-1) \& \text{BIT_INDEX_MASK}) < (\text{fromIndex} \& \text{BIT_INDEX_MASK}) \) (line 58) holds\(^{14}\), then the boolean \( \text{wordAligned} \) must be false (as \( (\text{fromIndex} \& \text{BIT_INDEX_MASK}) \) must be larger than 0). The get(int,int) method then uses \( \text{sourceIndex}+1 \) to access the \( \text{this.words} \) array. In this case, the bound is tighter, as \( \text{sourceIndex} + \text{targetWords} \) must now be smaller than \( \text{wordsInUse-1} \) (rather than just \( \text{wordsInUse} \)). 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 \( \text{result} \), the method calls the \( \text{recalculateWordsInUse()} \) method to establish the invariant in \( \text{result} \). The \( \text{wordsInUse} == 0 \) || \( \text{words[wordsInUse - 1]} != 0 \) and \( \text{wordsInUse} == \text{words.length} \) || \( \text{words[wordsInUse]} == 0 \) assertions from the class invariant may not be true for \( \text{result} \) when the method starts, as the method’s purpose is to establish the invariant. Specifically, \( \text{wordsInUse} \) may be too high, which is the case if \( \text{words[wordsInUse-1]} \) is zero. All other clauses from the class invariant hold when \( \text{recalculateWordsInUse()} \) is called. To restore the class invariant, the method lowers \( \text{wordsInUse} \) to the most significant element of \( \text{result.words} \) 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


A Annotated BitSet class

A.1 Internal fields of the class

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

```java
/*
   * 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;
```
22 /* Whether the size of "words" is user-specified. If so, we assume
23 * the user knows what he's doing and try harder to preserve it.
24 */
25 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) ) &&
  @ // 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
/*@ 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
/*@*/
```
A.3.3 recalculateWordsInUse()

Listing 32: The annotated recalculateWordsInUse() method.

A.3.4 The public BitSet constructors

Listing 33: The public BitSet constructors, with both public and private constructors.
public BitSet() {
    initWords(BITS_PER_WORD);
    sizeIsSticky = false;
}

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];
}

Listing 34: The annotated ensureCapacity(int) method.
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.
 */
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.
 */
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 true.
 * @param bitIndex a bit index
 * @throws IndexOutOfBoundsException if the specified index is negative
 * @since 1.0
 */
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 false.
 * @since 1.4
 */
public void clear() {
    /*@
    @  maintaining wordsInUse <= words.length;
    @  maintaining (forall bigint i; wordsInUse <= i < words.length; words[i] == 0);
    @*/
    /*@
    @  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);
        @*/
    }
A.3.10 \texttt{get(int,int)}

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

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

A.3.10 \texttt{get(int,int)}

\@* \texttt{get(int fromIndex, int toIndex)} {
  \@*
  \@* /\*\*
  \@*
  \@* Returns a new \texttt{@code BitSet} composed of bits from this \texttt{@code BitSet}
  \@* from \texttt{@code fromIndex} (inclusive) to \texttt{@code toIndex} (exclusive).
  \@* 
  \@* @param fromIndex index of the first bit to include
  \@* @param toIndex index after the last bit to include
  \@* @return a new \texttt{@code BitSet} from a range of this \texttt{@code BitSet}
  \@* @throws IndexOutOfBoundsException if \texttt{@code fromIndex} is negative,
  \@* or \texttt{@code toIndex} is negative, or \texttt{@code fromIndex} is
  \@* larger than \texttt{@code toIndex}
  \@* @since 1.4
  \@* */
  \@* 
  \@* normal behaviour
  \@* requires fromIndex >= 0 && fromIndex <= toIndex;
  \@* ensures \texttt{@result} != this && \invariant_for(\texttt{\result});
  \@* ensures (\forall \texttt{bigint i}; 0 <= i < \texttt{\result.wordsToSeq().length \therefore wordsToSeq()}[fromIndex + i]
  \@* : 0) == \texttt{\result.wordsToSeq()[i]};
  \@* ensures (\texttt{\result.wordsToSeq().length < (toIndex-fromIndex)}) =>
  \@* (\forall \texttt{bigint i}; \texttt{\result.wordsToSeq().length <= i < (toIndex-fromIndex)};
  \@* (fromIndex + i < \texttt{wordsToSeq().length \therefore wordsToSeq()}[fromIndex + i]
  \@* : 0) == 0);
  \@* assignable \texttt{nothing};
  \@*/
  \@*
  public \texttt{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 \texttt{BitSet(0)};
    \@*
    \@* Our suggested bug fix: */
    \@*
    \if (len < 0)
      len = \texttt{Integer.MAX_VALUE};
    \@*
    \@* An optimization
    \if (toIndex > len)
      toIndex = len;
    \@*
    \@*
    \BitSet result = new \texttt{BitSet(toIndex - fromIndex)};
    \int targetWords = wordIndex(toIndex - fromIndex - 1) + 1;
    \int sourceIndex = wordIndex(fromIndex);
    \texttt{boolean wordAligned} = ((fromIndex & BIT_INDEX_MASK) == 0);
    \@*
    \@* Process all words but the last word
    \@*
    \@* 
    \@* // Adjusting wordsToSeq for result:
    \@*
    \@* maintaining (\forall \texttt{bigint j}; 0 <= j < ((\texttt{bigint)i*(((bigint)BITS_PER_WORD)});
A.3.11 length()

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

```java
/*@
normal_behavior
@ requires true;
@ ensures \result \geq 0 \implies \result == Integer.MIN_VALUE;
@ ensures \result \neq 0 ==> \wordsToSeq().length - \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 < \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.

```java
// Our method for converting the actual representation to the logical representation.
/*@ private model strictly_pure \seq wordsToSeq() {
  \return \(\seq_def \bigint i; 0; (\bigint)\text{wordsInUse}*(\bigint)\text{BITS_PER_WORD};
  \(\text{words}[i / \text{BITS_PER_WORD}] >> (\int)(i \% \text{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.

```java
/**
 * 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.
 * More precisely,
 * <br>\{code BitSet.valueOf(longs).get(n) == ((longs[n/64] & (1L<<(n%64))) != 0)}
 * for all {code n < 64 * longs.length}.
 * 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.

```java
/**
 * Returns a new long array containing all the bits in this bit set.
 * More precisely, if
 * {
 *   long[] longs = s.toLongArray();
 *   for all 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.

```latex
\begin{verbatim}
// 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 )  
};
\end{verbatim}
```

B.2 `orJLongDef`

Listing 45: The `orJLongDef` rule.

```latex
\begin{verbatim}
// 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 )  
};
\end{verbatim}
```
B.3 PowTwoNeqZero

Listing 46: The PowTwoNeqZero rule.

1  PowTwoNeqZero {
2      \schemaVar \term int i;
3      \assumes( i >= 0 ==> )
4      \find( pow(2, i % 64) )
5      \sameUpdateLevel
6      \add( pow(2, i%64) != 0 ==> )
7      \heuristics( userTaclets1 )
8  };

B.4 PowTwoGreZero

Listing 47: The PowTwoGreZero rule.

1  PowTwoGreZero {
2      \schemaVar \term int i;
3      \assumes( i >= 0, i%64 < 63 ==> )
4      \find( pow(2, i % 64) )
5      \sameUpdateLevel
6      \add( pow(2, i % 64) >= 0 & inLong(pow(2, i % 64)) ==> )
7      \heuristics( userTaclets1 )
8  };

B.5 ModPowTwoNeqZero

Listing 48: The ModPowTwoNeqZero rule.

1  ModPowTwoNeqZero {
2      \schemaVar \term int i;
3      \assumes( i >= 0 )
4      \find( moduloLong(pow(2, i % 64)) )
5      \sameUpdateLevel
6      \add( moduloLong(pow(2, i%64)) != 0 ==> )
7      \heuristics( userTaclets1 )
8  };

B.6 ModPowTwoGreZero

Listing 49: The ModPowTwoGreZero rule.

1  ModPowTwoGreZero {
2      \schemaVar \term int i;
3      \assumes( i >= 0, i%64 < 63 ==> )
4      \find( moduloLong(pow(2, i % 64)) )
5      \sameUpdateLevel
6      \add( moduloLong(pow(2, i%64)) >= 0 & inLong(moduloLong(pow(2, i % 64))) ==> )
7      \heuristics( userTaclets1 )
8  };

B.7 orLongZero
Listing 50: The orLongZero rule.

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

B.8 binaryOrSingleBit

Listing 51: The binaryOrSingleBit rule.

```latex
\begin{verbatim}
// We set a single bit in x using binaryOr.
/binaryOrSingleBit {
.schemaVar \term int x;
.schemaVar \term int i;
\find (inLong(x), inInt(i), i >= 0 ==> 
.binaryOr(x, moduloLong(shiftleft(1, i))) ==>
.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 ==>)
// This is also only the case iff i = 63, so we add this to the list of assumptions.
"Set the sign bit": \replacewith (long_MIN + x) \add(\unsignedShiftrightJlong(x, i)\%2 = 0, 
.x >= 0, inLong(long_MIN + x), i = 63 ==>)
\heuristics (userTaclets2)
}
\end{verbatim}
```

B.9 unsignedShiftRightJlongDef

Listing 52: The unsignedShiftRightJlongDef rule.

```latex
\begin{verbatim}
// UNSIGNED shift right long:
\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)
}
```

45
B.10 \textbf{handleSignSHRLong}

Listing 53: The $\texttt{handleSignSHRLong}$ rule.

\begin{verbatim}
1 handleSignSHRLong {
2    \schemaVar \term int x;
3    \schemaVar \term int j;
4    \assumes (inInt(j), j >= 0, j <= 63, inLong(x), x >= 0 ==>)
5    \find ( unsignedshiftrightJlong(long_MIN + x, j) )
6    \sameUpdateLevel
7    // We know that the sign bit has been set -> the bit = 0.
8    "j = 63": \replacetaclet( 1 ) \add (j = 63 ==>)
9    // The sign bit has been set, but we not isolating the sign bit -> no change
10   // compared to the original value.
11   "j < 63": \add ( unsignedshiftrightJlong(long_MIN + x, j) % 2 = unsignedshiftrightJlong(x, j) % 2, j < 63 ==>)
12  \heuristics ( userTaclets2 )
13};
\end{verbatim}

B.11 \textbf{handleUnSHRlong}

Listing 54: The $\texttt{handleUnSHRlong}$ rule.

\begin{verbatim}
1 handleUnSHRlong {
2    \schemaVar \term int x;
3    \schemaVar \term int i;
4    \schemaVar \term int j;
5    \assumes (inLong(x), inLong(x + pow(2, i)), inInt(i), i >= 0, i < 63, inInt(j),
6      j >= 0, j <= 63, unsignedshiftrightJlong(x, i)%2 = 0 ==>)
7    \find ( unsignedshiftrightJlong(x + pow(2, i), j) )
8    \sameUpdateLevel
9    // With the SHR and the AND later, we will 'forget' the set bit.
10   "i != j": \add ( unsignedshiftrightJlong(x + pow(2, i), j) % 2 = unsignedshiftrightJlong(x, j) % 2, i != j ==>)
11   // We are looking at the set bit -> it will be set to 1.
12   "i = j": \add ( unsignedshiftrightJlong(x + pow(2, i), j) % 2 = 1, i = j ==>)
13  \heuristics ( userTaclets2 )
14};
\end{verbatim}