International Journal on Software Tools for Technology Transfer (2022) 24:783–802
https://doi.org/10.1007/s10009-022-00679-7

GENERAL

Special Issue: TACAS 2020

Verifying OpenJDK's `LinkedList` using KeY (extended paper)

Hans-Dieter A. Hiep[1,2] · Olaf Maathuis[4] · Jinting Bian[1,2] · Frank S. de Boer[1,2] · Stijn de Gouw[1,3]

Accepted: 29 August 2022 / Published online: 17 October 2022

Abstract
As a particular case study of the formal verification of state-of-the-art, real software, we discuss the specification and verification of a corrected version of the implementation of a linked list as provided by the Java Collection Framework.

Keywords Deductive verification · KeY · Java Modeling Language · Case study · Standard library · Bug

1 Introduction

Software libraries are the building blocks of millions of programs, and they run on the devices of billions of users every day. Therefore, their correctness is of the utmost importance. The importance and potential of formal software verification as a means of rigorously validating state-of-the-art, real software and improving it, is convincingly illustrated by its application to TimSort, the default sorting library in many widely used programming languages, including Java and Python, and platforms like Android (see [1,2]): a crashing implementation bug was found. The Java implementation of TimSort belongs to the Java Collection Framework which provides implementations of basic data structures and is among the most widely used libraries. Nonetheless, over the years, 877 bugs in the Collections Framework have been reported in the official OpenJDK bug tracker.

Due to the intrinsic complexity of modern software, the possibility of interventions by a human verifier is indispensable for proving correctness. This holds in particular for the Java Collection Framework, where programs are expected to behave correctly for inputs of arbitrary size. As a particular case study, we discuss the formal verification of a corrected version of the implementation of a linked list as specified by the class `LinkedList` of the Java Collection Framework in Java 8. Apart from the fact that the data structure of a linked list is one of the basic structures for storing and maintaining unbounded data, this is an interesting case study because it provides further evidence that formal verification of real software can lead to major improvements and correctness guarantees.

`LinkedList` is the only `List` implementation in the Collection Framework that allows collections of unbounded size. We found out that the Java linked list implementation does not correctly take into account the Java integer overflow semantics. It is exactly for large lists ($\geq 2^{31}$ items), that the implementation breaks in many interesting ways and sometimes even oblivious to the client. This basic observation gave rise to a number of test cases which show that Java's `LinkedList` class breaks the contract of 22 methods out of a total of 25 methods of the `List` interface!

In this paper, we focus on the integer overflow bug that we have discovered and on specifying and verifying an improved version of the `LinkedList` class which avoids this bug. We follow the general workflow (depicted in Fig. 1) that also underlies the TimSort case study. The workflow starts with a formalisation of the informal documentation of the Java

✉ Hans-Dieter A. Hiep
hdh@cwi.nl

Olaf Maathuis
olaf.maathuis@achmea.nl

Jinting Bian
j.bian@cwi.nl

Frank S. de Boer
frb@cwi.nl

Stijn de Gouw
sdg@ou.nl

1 Centrum Wiskunde & Informatica (CWI), Amsterdam, the Netherlands

2 Leiden Institute of Advanced Computer Science (LIACS), Leiden, the Netherlands

3 Open University (OU), Heerlen, the Netherlands

4 Achmea, Zeist, the Netherlands

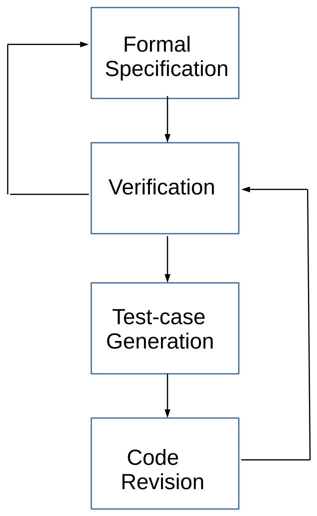

Fig. 1 Workflow

source code in the Java Modeling Language [3,4]. This formalisation goes hand in hand with the formal verification process: failed verification attempts can provide information about further refinements of the specifications. A failed verification attempt may also indicate an error in the code, and can as such be used for the generation of test cases to detect the error at run-time.

For our case study we use the state-of-the-art KeY theorem prover [5], because it fully formalizes the integer overflow semantics of Java and it allows to directly "load" Java programs. An archive of proof files and the KeY versions used in this study is available on-line in the Zenodo repository [6].

This paper is an extended version of the earlier publication [7]. The main additions in this extended paper are in-depth details about the integer overflow bug and its mitigation, improvements to the specification to include properties needed for client-side reasoning, and more details of the verification effort that also allows for reproduction of the proofs using a later version of KeY. In particular, in this paper, we now also specify properties of the `LinkedList` that are needed for reasoning about the correctness of clients of the linked list: clients of the linked list depend on properties that express what the contents of the linked list are, and may be oblivious to its internal structure. However, properties specifying the internal structure of the linked list are needed for showing the absence of the integer overflow bug: otherwise, client-side reasoning becomes unsound.

Plan of the paper Section 2 describes the `LinkedList` class in the Java Collection Framework, explaining the origin of the integer overflow bug and details on the reproduction and mitigation thereof. Section 3 explains our methodology, the reasons for choosing KeY as a verification system, and the false dichotomy between non-functional and functional verification. We describe our efforts of formally specifying and verifying the correctness of an adapted `LinkedList` class in Sect. 4, which shows the absence of the integer overflow. Section 5 discusses the main challenges posed by this case study and related work.

Impact As our case study involves real software, we have performed an analysis of existing Java code to estimate the use of `LinkedList` and the potential impact of a bug. A basic analysis of at least 140,000 classes[1], found by sampling Java packages on Maven Central and in software distributions such as Red Hat Linux, reveals 1677 cases of direct use of the `LinkedList` class where a constructor of `LinkedList` is invoked. As an over-estimation of the potential reach of such instances, we find at least 37,000 usage call sites where some method of the `Collection`, `List` or `Queue` interface is called. It is infeasible for us to analyze for each constructor call site where its resulting instance will be used. However, some usage of the `LinkedList` class occurs in potentially security-sensitive contexts, such as the library for checking certificates used by the Java secure socket implementation and checking the authenticity of the source of dynamically loaded bytecode.

We filed a security bug report to Oracle's security team, which was confirmed on October 31st, 2019. There was some communications back and forth: first our bug report was classified as a security-in-depth issue on November 7th, 2019, and later we received the notification that Oracle classified the issue as not-a-security issue, but as a functional issue (August 22nd, 2020):

> The bug is functional and doesn't cause any security issue for the JDK.
> We have examined the JDK and none of the instances of `LinkedList` would cause a security issue under the size overflow described. The class does not behave per its specification but we view this as a functional bug which should be reported [in the regular Java bug tracker].
> For an application to possibly have a security issue, it would need to use the `LinkedList` in some sensitive fashion. The number of elements of the `LinkedList` would need to be directly affected by an attacker's action and the billions of operations to drive the `LinkedList` into the overflow region would have to go unnoticed. At that point the unexpected exceptions would occur, but a well written program would handle those gracefully (since runtime exceptions can occur at any time). The index operations would fail but other operations would continue to work as expected. Uses of `LinkedList` that didn't use the indexed methods

[1] We have used the Rascal programming language [8] for loading `.jar`-files and analyzing their class file contents that revealed method call sites. We thank Thomas Degueule for his help setting up this experiment.

would never notice an issue. Additionally, traversing a large LinkedList would be too slow for most applications to consider using one.

Although it is true that billion operations have to go through unnoticed, it is a realistic scenario when an attacker can run code on a machine that processes big data, i.e. has a large available memory. Not all such operations, especially on Collection instances, are logged or can be detected if the attack is spreading the operations over some larger amount of time. Large time and memory requirements have not been a hindrance in exploits of the past, e.g. to escape the Java sandbox [9]. For example, adversarial clients in a cloud computing environment could perform attacks-from-within. Also, contrary to what is claimed above, there are possible runs where no exception at all is thrown, breaking invariants of client code that goes undetected.

We (twice) submitted a functionality bug report to the regular Java bug tracker but never received any confirmation of its reception: it remains unknown to us whether this latter bug report is received or any action is taken upon it. It seems good practice not to publicly acknowledge some bug reports, since the bug tracker may otherwise be used as a high-quality source of potential vulnerabilities [10]. Although our first security bug report was finally classified as not-a-security issue, we have, in the mean time, observed several issues[2], that show a pattern in which systematically the usage of LinkedList is eliminated from JDK's source code by replacing it with ArrayList. There are many possible explanations why such pattern of elimination has occurred, e.g. to improve run-time efficiency, and this pattern is not necessarily related to our bug report.

2 LinkedList in OpenJDK

LinkedList was introduced in Java version 1.2 as part of the Java Collection Framework in 1998. Fig. 2 shows how LinkedList fits in the type hierarchy of this framework: LinkedList implements the List interface, and also supports all general Collection methods as well as the methods from the Queue and Deque interfaces. The List interface provides positional access to the elements of the list, where each element is indexed by Java's primitive int type.

The structure of the LinkedList class is shown in Listing 1. This class has three attributes: a size field, which stores the number of elements in the list, and two fields that store a reference to the first and last node. Internally, it uses the private static nested Node class to represent the items in the list. A static nested private class behaves like

[2] JDK-8246048, JDK-8253178, JDK-8253179, JDK-8263561

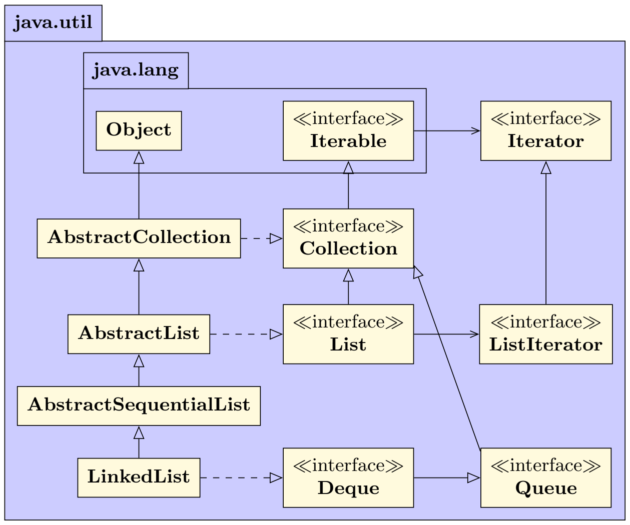

Fig. 2 LinkedList within Java Collections framework

```
public class
↪   LinkedList<E>
      extends
        ↪   AbstractSequentialList<E>
      implements
        ↪   List<E>,
        ↪   Deque<E>,
        ↪   ... {
  transient int size
  ↪   = 0;
  transient Node<E>
  ↪   first;
  transient Node<E>
  ↪   last;
  private static
  ↪   class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
    Node(Node<E> p, E
    ↪   i, Node<E> n)
    ↪   ...
  }
  ...
}
```

```
public boolean add(E e) {
  linkLast(e);
  return true;
}
void linkLast(E e) {
  final Node<E> l = last;
  final Node<E> newNode =
      new Node<>(l, e, null);
  last = newNode;
  if (l == null) first = newNode;
  else l.next = newNode;
  size++;
  modCount++;
}
```

Listing 1: The LinkedList class defines a doubly-linked list data structure.

a top-level class, except that it is not visible outside the enclosing class (LinkedList, in this case). Nodes are doubly linked; each node is connected to the preceding (field prev) and succeeding node (field next). These fields contain null in case no preceding or succeeding node exists. The data itself is contained in the item field of a node.

LinkedList contains 57 methods. Due to space limitations, we now focus on three characteristic methods: see Listing 1 and Listing 2. Method add(E) calls method linkLast(E), which creates a new Node object to store the new item and adds the new node to the end of the list. Finally the new size is determined by unconditionally incrementing the value of the size field, which has type int. Method indexOf(Object) returns the position (of type

```java
public int indexOf(Object o) {
    int index = 0;
    if (o == null) {
        for (Node<E> x = first; x != null; x = x.next) {
            if (x.item == null)
                return index;
            index++;
        }
    } else {
        for (Node<E> x = first; x != null; x = x.next) {
            if (o.equals(x.item))
                return index;
            index++;
        }
    }
    return -1;
}
```

Listing 2: The `indexOf` method searches for an element from the first node on.

`int`) of the first occurrence of the specified element in the list, or -1 if it's not present.

Each linked list consists of a sequence of nodes. Sequences are finite, indexing of sequences starts at zero, and we write $\sigma[i]$ to mean the ith element of some sequence σ. A *chain* is a sequence σ of nodes of length $n > 0$ such that: the `prev` reference of the first node $\sigma[0]$ is `null`, the `next` reference of the last node $\sigma[n-1]$ is `null`, the `prev` reference of node $\sigma[i]$ is node $\sigma[i-1]$ for every index $0 < i < n$, and the `next` reference of node $\sigma[i]$ is node $\sigma[i+1]$ for every index $0 \leq i < n-1$. The `first` and `last` references of a linked list are either both `null` to represent the *empty* linked list, or there is some chain σ between the `first` and `last` node, viz. $\sigma[0] =$ `first` and $\sigma[n-1] =$ `last`. Figure 3 shows example instances. Also see standard literature such as Knuth's [11, Section 2.2.5].

We make a distinction between the *actual* size of a linked list and its *cached* size. In principle, the size of a linked list can be computed by walking through the chain from the `first` to the `last` node, following the `next` reference, and counting the number of nodes. For performance reasons, the Java implementation also maintains a cached size. The cached size is stored in the linked list instance.

Two basic properties of doubly-linked lists are *acyclicity* and *unique first and last nodes*. Acyclicity is the statement that for any indices $0 \leq i < j < n$ the nodes $\sigma[i]$ and $\sigma[j]$ are different. First and last nodes are unique: for any index i such that $\sigma[i]$ is a node, the `next` of $\sigma[i]$ is `null` if and only if $i = n-1$, and `prev` of $\sigma[i]$ is `null` if and only if $i = 0$. Each item is stored in a separate node, and the same item may be stored in different nodes when duplicate items are present in the list.

2.1 Integer overflow bug

The size of a linked list is encoded by a signed 32-bit integer (Java's primitive `int` type) that has a two's complement binary representation where the most significant bit is a sign bit. The values of `int` are bounded and between -2^{31} (`Integer.MIN_VALUE`) and $2^{31}-1$ (`Integer.MAX_VALUE`), inclusive. Adding one to the maximum value, $2^{31}-1$, results in the minimum value, -2^{31}: the carry of addition is stored in the sign bit, thereby changing the sign.

Since the linked list implementation maintains one node for each element, its size is implicitly bounded by the number of node instances that can be created. Until 2002, the JVM was limited to a 32-bit address space, imposing a limit of 4 gigabytes (GiB) of memory. In practice this is insufficient to create 2^{31} node instances. Since 2002, a 64-bit JVM is available allowing much larger amounts of addressable memory. Depending on the available memory, in principle it is now possible to create 2^{31} or more node instances. In practice such lists can be constructed today on systems with 64 gigabytes of memory, e.g., by repeatedly adding elements. However, for such large lists, at least 20 methods break, caused by signed integer overflow. For example, several methods crash with a run-time exception or exhibit unexpected behavior!

Integer overflow bugs are a common attack vector for security vulnerabilities: even if the overflow bug may seem benign, its presence may serve as a small step in a larger attack. Integer overflow bugs can be exploited more easily on large memory machines used for 'big data' applications. Already, real-world attacks involve Java arrays with approximately $2^{32}/5$ elements [9, Section 3.2].

The `Collection` interface allows for collections with over `Integer.MAX_VALUE` elements. For example, its documentation (Javadoc) explicitly states the behavior of the `size()` method: 'Returns the number of elements in this collection. If this collection contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`'. The special case ('more than ...') for large collections is necessary because `size()` returns a value of type `int`.

When `add(E)` is called and unconditionally increments the `size` field, an overflow happens after adding 2^{31} elements, resulting in a negative `size` value. In fact, as the Javadoc of the `List` interface describes, this interface is based on integer indices of elements: 'The user can access elements by their integer index (position in the list), ...'. For elements beyond `Integer.MAX_VALUE`, it is very unclear what integer index should be used. Since there are only 2^{32} different integer values, at most 2^{32} node instances can be associated with an unique index. For larger lists, elements cannot be uniquely addressed anymore using an integer index. In essence, as we shall see in more detail below, the bounded nature of the 32-bit integer indices implies that the design of the `List` interfaces breaks down for large lists on 64-bit architectures. The above observations have many ramifications: it can be shown that 22 of 25 methods in the `List` interface are broken. Remarkably, the actual size of

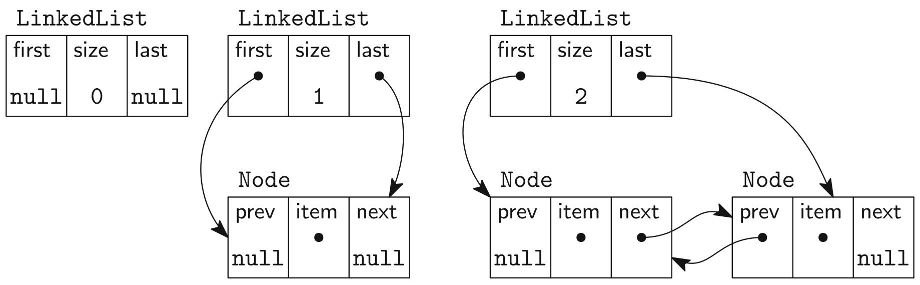

Fig. 3 Three example linked lists: empty, with a chain of one node, and with a chain of two nodes. Items themselves are not shown

the linked list remains correct as the chain is still in place: most methods of the `Queue` interface still work.

2.2 Reproduction

We have run a number of test cases to show the presence of bugs caused by the integer overflow.

The running Java version was Oracle's JDK8 (build 1.8.0 201-b09) that has the same `LinkedList` implementation as in OpenJDK8. Before running a test case, we set up an empty linked list instance. Below, we give an high-level overview of the test cases. Each test case uses `letSizeOverflow()` or `addElementsUntilSizeIs0()`: these repeatedly call the method `add()` to fill the linked list with `null` elements, and the latter method also adds a last element (`"this is the last element"`) causing `size` to be 0 again.

1. Directly after `size` overflows, the `size()` methods returns a negative value, violating what the corresponding Java documentation stipulates: its value should remain `Integer.MAX_VALUE` $= 2^{31} - 1$.

```
letSizeOverflow();
System.out.println("linkedList.size() = " +
↪  linkedList.size() + ", actual: " + count);
// linkedList.size() = -2147483648, actual:
↪  2147483648
```

   Clearly this behavior is in contradiction with the documentation. The actual number of elements is determined in the test case by having a field `count` (of type `long`) that is incremented each time the method `add()` is called.
2. The query method `get(int)` returns the element at the specified position in the list. It throws an `IndexOutOfBoundsException` exception when `size` is negative. From the informal specification, it is unclear what indices should be associated with elements beyond `Integer.MAX_VALUE`.

```
letSizeOverflow();
System.out.println(linkedList.get(0));
// Exception in thread "main"
↪  IndexOutOfBoundsException: Index: 0, Size:
↪  -2147483648
//         at java.util.LinkedList.checkElementIndex
// (LinkedList.java:555) ...
```

3. The method `toArray()` returns an array containing all of the elements in this list in proper sequence (from first to last element). When `size` is negative, this method throws a `NegativeArraySizeException` exception. Furthermore, since the array size is bounded by $2^{31} - 1$ elements[3], the contract of `toArray()` is unsatisfiable for lists larger than this. The method `Collections.sort(List` $<$ `T` $>$`)` sorts the specified list into ascending order, according to the natural ordering of its elements. This method calls `toArray()`, and therefore also throws a `NegativeArraySize Exception`.

```
letSizeOverflow();
Collections.sort(linkedList);
// Exception in thread "main" NegativeArraySizeException
//    at
↪  java.util.LinkedList.toArray(LinkedList.java:1050)...
```

4. Method `indexOf(Object o)` returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. However due to the overflow, it is possible to have an element in the list associated to index -1, which breaks the contract of this method.

```
addElementsUntilSizeIs0();
String last;
System.out.println("linkedList.getLast() = " + (last
↪  = linkedList.getLast()));
// linkedList.getLast() = This is the last element
System.out.println("linkedList.indexOf(" + last + ")
↪  = " + linkedList.indexOf(last));
// linkedList.indexOf(This is the last element) = -1
```

5. Method `contains(Object o)` returns true if this list contains the specified element. If an element is associated with index -1, it will indicate wrongly that this particular element is not present in the list.

```
addElementsUntilSizeIs0();
String last;
System.out.println("linkedList.getLast() = " + (last
↪  = linkedList.getLast()));
// linkedList.getLast() = This is the last element
System.out.println("linkedList.contains(" + last + ")
↪  = " linkedList.contains(last));
// linkedList.contains(This is the last element) =
↪  false
```

[3] In practice, the maximum array length turns out to be $2^{31} - 5$, as some bytes are reserved for object headers, but this may vary between Java versions [9,12].

6. Method `descendingIterator()` returns an instance of `DescendingIterator` (line 3 on the left of Listing 3). It asserts `isPositionIndex(index)`, meaning that `index` $>=$ 0&&`index` $<=$ `size` holds. In case `size` overflows and becomes negative, it is clear that this inequality is false (regardless of whether there are actually more items to iterate over), and as a result a `NoSuchElement Exception` exception is thrown (line 20 on the right) when attempting to obtain the next element.

```
letSizeOverflow();
Iterator<String> it =
↪  linkedList.descendingIterator();
String s = it.next();
//  java.util.NoSuchElementException
//     at java.util.LinkedList$ListItr.previous
//  (LinkedList.java:905)
//     at java.util.LinkedList$DescendingIterator.next
// (LinkedList.java:998)
```

7. Method `spliterator()` creates a `Spliterator` over the elements in this list. See Listing 4. The constructor sets the field `est`, that represents the estimated size, to -1 (line 2), but when the field is needed it takes the value of `size` of `LinkedList` (line 24). When `size` < 0 due to an overflow, `getEst()` will return a negative value (line 32) which will cause `trySplit()` to erroneously return null (line 37).

```
letSizeOverflow();
Spliterator<String> s = linkedList.spliterator();
Spliterator<String> t = s.trySplit();
System.out.println("t == " + t);
//  t == null
```

8. Method `isEmpty()`, which is inherited from class `AbstractCollection`, returns true if this collection contains no elements (and false otherwise), according to the Java documentation of this method of interface `Collection` which is implemented by `AbstractCollection`.

```
addElementsUntilSizeIs0();
System.out.println("linkedList.isEmpty(): " +
↪  linkedList.isEmpty());
// linkedList.isEmpty(): true
```

The `isEmpty()` method just returns `size() == 0`. However, since `size()` itself does not conform to its accompanying Java documentation (see test case 1), the outcome of `isEmpty()` is faulty.

9. Method `subList(int fromIndex, int toIndex)` is inherited from class `AbstractList`. The Java documentation states: “Returns a view of the portion of this list between the specified `fromIndex`, inclusive, and `toIndex`, exclusive.”, so `subList(0,0)` must give an empty list. Instead, an `IndexOutOf BoundsException` exception is thrown.

```
letSizeOverflow();
linkedList.subList(0, 0);
//  java.lang.IndexOutOfBoundsException: toIndex = 0
```

```
 1  public Iterator<E>
 2    descendingIterator()
      ↪ {
 3    return new
      ↪ Descending
 4    Iterator();
 5  }
 6  private class
    ↪ DescendingIterator
 7    implements
      ↪ Iterator<E> {
 8    private final
      ↪ ListItr itr =
 9      new ListItr
10      (size());
11    public boolean
      ↪ hasNext() {
12      return itr.has
13      Previous();
14    }
15    public E next() {
16      return
        ↪ itr.previous();
17    }
18    public void
      ↪ remove() {
19      itr.remove();
20    }
21  }
```

```
 1  private class ListItr
 2    implements ListIterator<E> {
 3    private Node<E> next;
 4    private int nextIndex;
 5    ...
 6    ListItr(int index) {
 7      // assert isPositionIndex
 8      (index);
 9      next = (index == size) ?
10       null :
11        node(index);
12      nextIndex = index;
13    }
14    ...
15    public boolean hasPrevious() {
16      return nextIndex > 0;
17    }
18    public E previous() {
19      ...
20      if (!hasPrevious())
21        throw
22          new NoSuchElement
23          Exception();
24      ...
25      }
26    ...
27    }
```

Listing 3: DescendingIterator. This class malfunctions when an instance is created and `size` has overflowed to a negative value.

```
//    at java.util.SubList.<init>(AbstractList.java:622)
//    at
↪  java.util.AbstractList.subList(AbstractList.java:484)
```

Specifically, method `letSizeOverflow()` adds 2^{31} elements that causes the overflow of `size`. Method `add ElementsUntilSizeIs0()` first adds $2^{32} - 1$ null values, after which `size` becomes -1. Then, it adds a final non-null value so that `size` is 0 again. All elements added are null, except for the last element. For test cases 4 and 5, we deliberately exploit the overflow bug to associate an element with index -1. This means that method `indexOf(Object)` for this element returns -1, which according to the documentation means that the element is not present.

For test cases 1, 2, 3, 6, 7 and 9 (the *small* test cases) we needed 65 gigabytes of memory for the JRE on a VM with 67 gigabytes of memory. For test cases 4, 5 and 8 (the *large* test cases) we needed 167 gigabytes of memory for the JRE on a VM with 172 gigabytes of memory. All test cases were executed on a super computer of a private cloud (SURFsara).

The minimal amount of memory for triggering the bug is 64 gigabyte of memory. The reason why this amount cannot be reduced is quite intricate, but interesting enough to warrant the following discussion. Java eventually stores objects in physical memory that can be addressed by either 32-bit or 64-bit pointers, depending on the underlying architecture. As we already discussed before, the overflow bug does not occur on 32-bit architectures. However, on 64-bit architectures the

```java
public Spliterator<E> spliterator() {
  return new LLSpliterator<E>(this, -1, 0);
}
static final class LLSpliterator<E> implements Spliterator<E>
↪  {
  ...
  final LinkedList<E> list; // null OK unless traversed
  Node<E> current; // current node; null until initialized
  int est; // size estimate; -1 until first needed
  int expectedModCount; // initialized when est set
  LLSpliterator(LinkedList<E> list, int est, int
  ↪  expectedModCount) {
    this.list = list;
    this.est = est;
    this.expectedModCount = expectedModCount;
  }
  final int getEst() {
    int s; // force initialization
    final LinkedList<E> lst;
    if ((s = est) < 0) {
      if ((lst = list) == null)
        s = est = 0;
      else {
        expectedModCount = lst.modCount;
        current = lst.first;
        s = est = lst.size;
      }
    }
    return s;
  }
  ...
  public Spliterator<E> trySplit() {
    Node<E> p;
    int s = getEst();
    if (s > 1 && (p = current) != null) {
      ...
      return Spliterators.spliterator(a, 0, j,
      ↪  Spliterator.ORDERED);
    }
    return null;
  }
  ...
}
```

Listing 4: The method `spliterator()` creates a `Spliterator` over the elements in this list.

```java
1  public abstract class AbstractList<E> extends
   ↪  AbstractCollection<E>
2    implements List<E> {
3    ...
4    public List<E> subList(int fromIndex, int toIndex) {
5      return (this instanceof RandomAccess ?
6        new RandomAccessSubList<>(this, fromIndex,
         ↪  toIndex) :
7        new SubList<>(this, fromIndex, toIndex));
8    }
9    ...
10 }
11 class SubList<E> extends AbstractList<E> {
12   ...
13   private int size;
14   SubList(AbstractList<E> list, int fromIndex, int
     ↪  toIndex) {
15     ...
16     if (toIndex > list.size())
17       throw new IndexOutOfBoundsException("toIndex = " +
         ↪  toIndex);
18     ...
19   }
20   ...
21 }
```

Listing 5: Creates an instance of (non-public) type `SubList` (line 7), and when the size is negative due to an overflow, an `IndexOutOfBoundsException` is thrown in its constructor (line 17).

Java virtual machine may sometimes still employ only 32-bit for storing an object address, by using a pointer compression technique and ensuring that objects in memory are aligned.

If objects are aligned on 8 byte boundaries, the three least significant digits of each 64-bit address is known to be 000 and thus can be truncated. One may still use a 32-bit address, but interpret it as a 64-bit memory location by multiplying its value by 8. Then the maximum amount of memory that a virtual machine can use is $2^{(32+3)}$, being 32 GiB. But, how many internal nodes in `LinkedList` can we fit in that amount? That depends on the size of each object. The size of an object consequently depends on an object header (which is 8 byte for storing a hash code and object locking data and 4 byte for a class pointer), and the fields (which is 4 byte per reference field if we compress addresses, and we have three fields for nodes): so for `Node` we require at least 24 bytes. But how many nodes then fit within 32GiB of memory? That is less than 2^{31}, so the overflow would never occur and instead an `OutOfMemoryError` happens first.

If, however, objects are aligned on 16 byte boundaries and we still use compressed 32-bit addresses (but multiplied by 16 to obtain the 64-bit memory location), then we need to add padding to each node object and that makes each node take up 32 byte of space. Now we could make use of $2^{(32+4)}$ bytes of memory, being 64 GiB. However, since every object now takes 32 bytes of space, this leaves us with at most 2^{31} node objects: but in practice we could never trigger the bug, as the Java heap also contains a number of other objects. So also here, we would trigger `OutOfMemoryError` first before the overflow happens.

So, for our small test cases, we let objects be aligned on 32 byte boundaries (since the `Node` is already 32-byte aligned, this does not cost more space than in the previous case) and use compressed 32-bit addresses. This allows a heap of up to $2^{(32+5)}$ bytes, being 128 GiB. Then by actually using a little bit more than 64 GiB of memory, we can trigger the overflow to happen: we could now construct 2^{31} internal node objects, and these objects can still be addressed by using 32-bit compressed addresses. However, in the large test cases, to reproduce the bug where the size would come back to zero again we need more memory. No longer can we use 32-bit compressed addresses, and now we need to make use of 64-bit addresses directly. That means then that no longer we can compress the addresses. So the total space required for an internal node becomes 40 bytes: 16 bytes for the header, and 8×3 bytes for the fields. Creating 2^{32} node objects then takes roughly 160 GiB of memory.

2.3 Mitigation

A mitigation of the problem described above is to change the linked list implementation to avoid the problems as encountered. There are different directions for mitigation, but there are different trade-offs to be made. One could compare each mitigation direction by the following aspects: its correctness with respect to the (informal) contract, compatibility with existing client code, completeness in the sense of being future-proof, and run-time efficiency.

We have considered the following directions for mitigating the overflow bug:

- *do not fix* but change the specification in the Java documentation to reflect the existing behavior,
- *fail fast* by raising a runtime exception as soon as an overflow is detected,
- *long cached size* avoids the overflow by keeping the actual size and the cached size in sync. However, the interface would still use `int` indices and not all objects can be retrieved by index. It avoids the overflow, since on a 64-bit JVM no more objects can exists than addresses for the machine are available, an `OutOfMemoryError` would occur before any overflow may happen;
- *long indices and size* change the interface and make the implementation no longer backwards compatible,
- *`BigInteger` indices and size* change the interface and ensure the implementation is future-proof in case the JVM is ported to machines with a word size even larger than 64-bit.

We first discuss an approach based on `BigIntegers`. Although this solution solves the issue, it is *not* compatible with the existing `List` and `LinkedList` types at the compilation level, as we shall see. We then describe the *fail fast* solution. At the compilation level, this latter solution is compatible with the existing `LinkedList` and thus can serve as a drop-in replacement.

BigInteger indices solution. To avoid overflow and boundedness problems, one may use the `BigInteger` or `long` type to refer to indices of elements and the `size` of the list. As long as the chosen integer type contains more values than the number of objects that can be created[4], all items in a list can be associated with a unique index. However, this approach does have a major drawback. It is impossible to retrofit such a class it in the existing Java Collections Framework: the class cannot implement the existing `List` interface because the signatures of the methods in that interface use `int` rather than `BigInteger`. Changing the `List` interface to use `BigInteger` would break all existing client classes that use `List` or any class that implements it (in the sense that those would not compile anymore!). Clearly this is a deal breaker, so this approach cannot supply a drop-in replacement of the existing `LinkedList`. Furthermore, changing

[4] This is bounded by the architecture on which the virtual machine runs. For current machines it is typically 32-bit or 64-bit.

```
public interface List<E> extends Collection<E> {
    BigInteger size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    ~~Object[] toArray();~~
    ~~<T> T[] toArray(T[] a);~~
    boolean add(E e);
    boolean remove(Object o);
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean addAll(BigInteger index, Collection<? extends
    ↪   E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    boolean equals(Object o);
    int hashCode();
    E get(BigInteger index);
    E set(BigInteger index, E element);
    void add(BigInteger index, E element);
    E remove(BigInteger index);
    BigInteger indexOf(Object o);
    BigInteger lastIndexOf(Object o);
    ListIterator<E> listIterator();
    ListIterator<E> listIterator(BigInteger BigInteger
    ↪   index);
    List<E> subList(BigInteger fromIndex, BigInteger
    ↪   toIndex);
}
```

Listing 6: List interface with BigInteger.

the argument from a primitive type `int` to a reference type such as `BigInteger` likely has a negative effect on the run-time performance.

To implement this solution, one could create a separate `List` type hierarchy, which, as stated, is incompatible with the existing collection framework type hierarchy. Listing 6 highlights the necessary changes for this new `List` interface. In addition to the highlighted changes, both overloaded methods `toArray()` can no longer be part of this interface, as these methods ought to return *all* elements of the collection, but arrays are inherently bounded (the declared maximum capacity when creating an array must be a primitive `int` integer, and as such, is bounded and the elements of an array are addressed by a primitive (bounded) `int` index).

The next step would be to implement a `LinkedList BigInteger` class, along the lines of Listing 7. Clients that (may) truly need unbounded linked lists, or lists with more than 2.147.483.647 items, should then use this class. Other index-based components of the Java collection framework would then have to be cloned too, e.g. `AbstractSequentialList`, `AbstractList`, `AbstractCollection`, `List`, `Collection`, and `Deque`. In addition, other classes of the collection framework like `Arrays`, `Collections`, `ArrayList`, `Map`, etc., need to be changed as well.

Fail fast solution. In this solution, we ensure that the overflow of `size` never occurs by using a form of failure atomicity. In particular, whenever an operation is performed that would cause the size to overflow, the operation instead throws an exception and leaves the list unchanged. An excep-

```
import java.math.BigInteger;
public class LinkedListBigInteger<E> {
  transient BigInteger size =
  ↪  BigInteger.ZERO;
  transient Node<E> first;
  transient Node<E> last;
  private static class Node<E> {...}
  public void add(BigInteger index, E
  ↪  element) {
    checkPositionIndex(index);
    if (index.equals(size))
    ↪  linkLast(element);
    else linkBefore(element, node(index));
  }
  Node<E> node(BigInteger index) {
    // assert isElementIndex(index);
    if
    ↪  (index.compareTo(size.divide(BigInteger.TWO))
    ↪  < 0) {
      Node<E> x = first;
      for (BigInteger i = BigInteger.ZERO;
      ↪  i.compareTo(index) < 0;
        i = i.add(BigInteger.ONE)) { x =
        ↪  x.next; } return x;
    } else {
      Node<E> x = last;
      for (BigInteger i =
      ↪  size.subtract(BigInteger.ONE);
      ↪  i.compareTo(index) > 0;
        i = i.subtract(BigInteger.ONE)) { x =
        ↪  x.prev; } return x;
    }
  }
  void linkLast(E e) {
    ... size = size.add(BigInteger.ONE); ...
  }
  void linkBefore(E e, Node<E> succ) {
    ... size = size.add(BigInteger.ONE); ...
  }
  private boolean isElementIndex(BigInteger
  ↪  index) {
    return index.compareTo(BigInteger.ZERO)
    ↪  >= 0 && index.compareTo(size) < 0;
  }
  private boolean isPositionIndex(BigInteger
  ↪  index) {
    return index.compareTo(BigInteger.ZERO)
    ↪  >= 0 && index.compareTo(size) <= 0;
  }
  private String outOfBoundsMsg(BigInteger
  ↪  index) {
    return "Index: "+index+", Size: "+size;
  }
  private void checkPositionIndex(BigInteger
  ↪  index) {
    if (!isPositionIndex(index))
      throw new
      ↪  IndexOutOfBoundsException(outOfBoundsMsg(index));
  }
  ...
}
```

Listing 7: BigInteger indices solution.

tion is triggered right before the overflow would otherwise occur, so the value of `size` is guaranteed to be bounded by `Integer.MAX_VALUE`, i.e. it never becomes negative.

This solution requires a slight adaptation of the implementation: for methods that increase the `size` field, only one additional check has to be performed before a `LinkedList` instance is modified. This checks whether the result of the method causes an overflow of the `size` field. Under this condition, an `Illegal-StateException` is thrown. Thus, only in states where `size` is less than `Integer.MAX_VALUE`, it is acceptable to add a single element to the list. This additional check may have a negative effect on the run-time efficiency of the code.

To summarize, this solution is closest to the original implementation, and does not break existing clients that use `LinkedList` as long as they use at most $2^{31} - 1$ elements of `LinkedList`. In contrast to the existing `LinkedList`, the behaviour of the methods is specified clearly and unambiguously for large collections.

We shall work in a separate class called `BoundedLinkedList`: this is the improved version that does not allow more than $2^{31} - 1$ elements. Compared to the original `LinkedList`, four methods are added:

```
private boolean isMaxSize() {
  return size == Integer.MAX_VALUE;
}
private void checkSize() {
  if (isMaxSize())
    throw new IllegalStateException("Not enough space");
}
private boolean enoughSpace(int s) {
  return 0 <= s && s <= Integer.MAX_VALUE - size();
}
private void checkSize(int s) {
  if (!enoughSpace(s))
    throw new IllegalStateException("Not enough space");
}
```

These methods implement an overflow check. The method `checkSize()` is called before any modification occurs that increases the size by one, and the method `checkSize(int)` is called before a modification occurs that increases the size by any non-negative quantitiy. These methods ensure that `size` never overflows. Some methods now differ when compared to the original `LinkedList`, as they involve an invocation of the `checkSize()` or `checkSize(int)` method.

3 Methodology

In this section we describe the overall methodology that underlies the specification and verification effort of our work. What is the purpose of formal specification and verification in the first place? Our goal is to show the absence of the integer overflow bug in the aforementioned fail fast solution. After this particular goal has been accomplished, what improvements can still be made that affects the specification and consequently the verification process?

The purpose of our specification and verification effort is to verify whether our formalization of the given informal Javadoc specifications (stated in natural language) of the `LinkedList` holds. Given that we already identified the overflow issue in the previous section, here we restrict our attention only to the revised `BoundedLinkedList`. Further, to simplify matters, we restrict ourselves only to the linked list and ignore the rest of the collection framework and other Java classes: methods that involve parameters of interface types, serialization, and reflection are considered out of scope.

We focus on two aspects of program correctness: functional and non-functional requirements. It is commonly understood that these two classes of requirements form a dichotomy, so that we can separate our concerns. On the one hand, functionality requirements are expressed by ascertaining an input/output relation, formulating in the assertion language how the output is obtained in terms of the input. On the other hand, non-functional requirements impose e.g. absence of crashing, termination of the program, or limitations on its effects. In our case study, we initially focused on the latter class, viz. establishing the absence of integer overflow. This suggests that we only need to formalize the non-functional requirements. Although the dichotomy between functional and non-functional requirements is clear superficially, it is a *false dichotomy* when one also considers establishing the veracity of formal specifications by the means of verification. Namely, to show, during the verification process, that a non-functional property holds (e.g. absence of crashing), it is often necessary to *also* specify additional functional properties (e.g. those required in class invariants and loop invariants).

We have clearly seen this false dichotomy as we followed along the workflow (depicted in Fig. 1), where we refined the specifications in multiple iterations, each time revisiting the formulation of specifications after getting stuck in the verification process. Each time we were stuck, it was necessary to formalize additional properties, e.g. describing the internal structure of the linked list. At one point in time, we were able to establish the absence of overflow for all our methods considered in scope. By taking this approach, we have thus specified certain aspects of the functionality of the program too, as could be seen from the class invariant we obtained at that point that clearly describes the internal structure of a linked list.

However, there was still room for improvement: we then shifted our focus on a different purpose of the formal specification. Namely, the properties needed from a implementation-side perspective are not sufficient for establishing properties from a client-side perspective. In the latter case one would also need a description of the *contents* of the linked list. In our last iteration of the workflow, we have refined the specification to furthermore include a description that relates the contents of the linked list before and after each method call. This allows for the verification of a client-side property that e.g. after adding objects by multiple calls to the `add(Object)` method, the `contains(Object)` must return true for precisely those objects that were added in the first place. Without specifying the relationship between the contents of the linked list before and after each method, such client-side property cannot be verified.

4 Specification and verification of `BoundedLinkedList`

`(Bounded)LinkedList` inherits from `AbstractSequentialList`, but we consider its inherited methods out of scope. These methods operate on other collections such as `removeAll` or `containsAll`, and methods that have other classes as return type such as `iterator`. However, the implementation of these inherited methods themselves call methods overridden by `(Bounded)LinkedList`, and can not cause an overflow by themselves.

We have made use of KeY's stub generator to generate dummy contracts for other classes that `BoundedLinkedList` depends on, such as for the inherited interfaces and abstract super classes: these contracts are the most general possible and specify that every method may arbitrarily change the heap. In other words, the generated stub contracts are satisfied by any method, no assumptions are made about the behavior of stub methods. This (conservative assumption) ensures that if a method that calls these stubs can be proven, this proof is sound. The stub generator moreover deals with generics, roughly by replacing generic type parameters with the `Object` type. For exceptions we modify their stub contract to assume that their constructors are *pure*, viz. leaving existing objects on the heap unchanged. An important stub contract is the equality method of the absolute super class `Object`, which we have adapted: we assume every object has a *side-effect free*, *terminating* and *deterministic* implementation of its equality method[5] and formalize this in the post-condition:

```
public class Object {
    /*@ public normal_behavior
      @   requires true;
      @   ensures \result == self.equals(param0);
      @*/
    public /*@ helper strictly_pure @*/ boolean
      equals(/*@ nullable */ Object param0);
    ...
}
```

4.1 Specification

Following our workflow, we have iterated a number of times before the specifications we present here were obtained. This is a costly procedure, as revising some specifications requires redoing most verification effort. Until sufficient information is present in the specification, proving for example termination of a method is difficult or even impossible: from partial,

[5] In reality, there are Java classes for which equality is not terminating normally, that is, with an infinite stack space the execution diverges. A nice example is `LinkedList` itself, where adding a list to itself leads to a `StackOverflowError` when testing equality with a similar instance. We consider the issue out of scope of this study as this stack overflow behavior is explicitly described by the Javadoc.

failed verification attempts, and the (formulas in) the corresponding open goal in KeY, one can get an intuitive idea of why a proof is stuck and revise the specification accordingly.

Ghost and model fields. We use JML's ghost fields and model fields: these are logical fields (a form of auxiliary variables) that associate a value to each object in the heap. These fields are only used for specification and verification purposes. The value of ghost fields is determined by set-statements that the user can add (in the form of JML annotations) inside method bodies, in all methods where the value should be updated. The value of a model field is determined by a single represents clause, that describes the value by its relation to the underlying fields (or ghost fields) of the class. At run-time, neither ghost fields nor model fields are present and cannot affect the course of execution. We annotate our improved class with two ghost fields: `nodeList`, `nodeIndex`. We also annotated our class with a model field: `itemList`.

The type of the `nodeList` ghost field is an abstract data type of sequences, a KeY built-in type. It has standard constructors and operations that can be used in contracts and in JML annotations. A sequence has a length, which is finite but unbounded. The type of a sequence's length is `\bigint`. In KeY a sequence is unityped: all its elements are of the *any* sort, which can be any Java object reference or primitive, or built-in abstract data type. One needs to apply appropriate casts and track type information for a sequence of elements in order to cast elements of the *any* sort to any of its subsorts. As such, we restrict `nodeList` to be a sequence of references to internal `Node` objects.

The `nodeIndex` ghost field is used as a ghost parameter with unbounded but finite integers as type. This ghost parameter is only used for specifying the behavior of the methods `unlink(Node)` and `linkBefore(Object, Node)`. The ghost parameter tracks at which index the `Node` argument is present in the `nodeList`. This information is implicit and not needed at run-time.

Although using ghost fields `nodeList` and `nodeIndex` is sufficient for showing the absence of the integer overflow, there is still room for improvement of the specification to make it suitable for client-side reasoning as well. The `itemList` is a model field that is directly related to `nodeList`: it is a sequence of the items (not wrapped in enclosing `Node` instances, in contrast to `nodeList`) stored in the linked list. This model field is important for clients that make use of linked lists: clients have no knowledge of the internal and encapsulated `Node` type, they are only interested in what items contained by the linked list and their order. This is exactly the information that `itemList` exposes. Listing 8 shows the relation between `itemList` and `nodeList`. In particular, the represents clause ensures that the value of the `itemList` is exactly the sequence of items that are currently stored in the item field of the `Node` objects in the `nodeList`.

```
//@ public model \seq itemList;
//@ represents itemList = (\seq_def \bigint i; 0;
↪  nodeList.length; ((Node)nodeList[i]).item);
```

Listing 8: Definition of the model field `itemList` in terms of the ghost field `nodeList`.

```
1   //@ private ghost \seq nodeList;
2   //@ private ghost \bigint nodeIndex;
3   /*@ invariant
4     @  nodeList.length == size &&
5     @  nodeList.length <= Integer.MAX_VALUE &&
6     @  (\forall \bigint i; 0 <= i < nodeList.length;
7     @      nodeList[i] instanceof Node) &&
8     @  ((nodeList == \seq_empty && first == null && last
      ↪  == null)
9     @   || (nodeList != \seq_empty && first != null &&
10    @        first.prev == null && last != null &&
11    @        last.next == null && first ==
      ↪  (Node)nodeList[0] &&
12    @        last == (Node)nodeList[nodeList.length-1]))
      ↪  &&
13    @  (\forall \bigint i; 0 < i < nodeList.length;
14    @     ((Node)nodeList[i]).prev ==
      ↪  (Node)nodeList[i-1]) &&
15    @  (\forall \bigint i; 0 <= i < nodeList.length-1;
16    @     ((Node)nodeList[i]).next ==
      ↪  (Node)nodeList[i+1]);
17    @*/
```

Listing 9: The class invariant expressed using the `nodeList` ghost field.

Class invariant The ghost field `nodeList` is used in the class invariant of the bounded linked list, see Listing 9. We relate the fields `first` and `last` that hold a reference to a `Node` instance, and the chain between `first` and `last`, to the contents of the sequence in the ghost field `nodeList`. This allows us to express properties in terms of `nodeList`, where they reflect properties about the chain on the heap. One may compare this invariant with the description of chains as given in Sect. 2. The actual size of a linked list is the length of the ghost field `nodeList`, whereas the cached size is stored in a 32-bit signed integer field `size`. On line 4, the invariant expresses that these two must be equal. Since the length of a sequence (and thus `nodeList`) is never negative, this implies that the size field never overflows. On line 5, this is made explicit: the real size of a linked list is bounded by `Integer.MAX_VALUE`. Line 5 is redundant as it follows from line 4, since a 32-bit integer never has a value larger than this maximum value. The condition on lines 6–7 requires that every node in `nodeList` is an instance of `Node` which implies it is non-`null`.

A linked list is either empty or non-empty. On line 8, if the linked list is empty, it is specified that `first` and `last` must be `null` references. On lines 9–12, if the linked list is non-empty, we specify that `first` and `last` are non-`null` and moreover that the `prev` field of the first `Node` and the `next`

```
1   // implements java.util.List.set
2   /*@
3     @ also
4     @ public normal_behavior
5     @   requires
6     @     0 <= index < nodeList.length;
7     @   ensures
8     @     ((Node)nodeList[index]).item == element &&
9     @     nodeList == \old(nodeList) &&
10    @     \result == \old(((Node)nodeList[index]).item);
11    @ public exceptional_behavior
12    @   requires
13    @     index < 0 || index >= nodeList.length;
14    @   signals_only IndexOutOfBoundsException;
15    @   signals (IndexOutOfBoundsException e) true;
16    @*/
17  public /*@ nullable @*/ Object
18  set(int index, /*@ nullable @*/ Object element) {
19      checkElementIndex(index);
20      Node x = node(index);
21      Object oldVal = x.item;
22      x.item = element;
23      return oldVal;
24  }
```

Listing 10: Method `set(int, Object)` annotated with JML.

field of the last `Node` are `null`. The `nodeList` must have as first element the node pointed to by `first`, and `last` as last element. In any case, but vacuously true if the linked list is empty, the `nodeList` forms a chain of nodes. Lines 13–16 capture this: for every node at index $0 < i <$ `size`, the `prev` field must point to its predecessor, and similar for successor nodes.

We note three interesting properties that are implied by the above invariant: acyclicity and unique first and last nodes. These properties can be expressed as JML formulas as follows:

```
(\forall \bigint i; 0 <= i < nodeList.length - 1;
    (\forall \bigint j; i < j < nodeList.length;
        nodeList[i] != nodeList[j])) &&
(\forall \bigint i; 0 <= i < nodeList.length;
    nodeList[i].next == null <==> i = nodeList.length - 1) &&
(\forall \bigint i; 0 <= i < nodeList.length;
    nodeList[i].prev == null <==> i = 0)
```

These properties are not literally part of our class invariant, but their validity is proven interactively in KeY as a consequence of the class invariant. Otherwise, we would need to reestablish also these properties each time we show the class invariant holds.

Methods. All methods within our scope are given a JML contract that specifies their normal behavior (i.e. the cases where the method terminates normally without an exception) and their exceptional behavior. As an example contract, consider the `lastIndexOf(Object)` method in Listing 12: it searches through the chain of nodes until it finds a node with an item equal to the parameter. This method is interesting due to a potential overflow of the result value.

Another concrete example that shows the use of our ghost field `nodeList` is the `set` method. The method

```
static boolean exampleClient(BoundedLinkedList ll, int
↪   n, int m) {
    Object o1 = ll.get(n);
    ll.set(m, new Object());
    Object o2 = ll.get(n);
    return o1 == o2;
}
```

Listing 11: A client of a linked list that calls `set(int, Object)`. If `n` and `m` are different but in-bound indices, then we expect that the result of the `get(int)` call after the `set(int, Object)` call results in the same item as the call before.

`set(int, Object)` replaces an item at the specified position with a given element, and returns to the caller the element that was originally present at that position. We here demonstrate how contracts have evolved over time following our workflow, and how we later discovered that `itemList` is needed in the specification. Listing 10 shows the method `set(int, Object)` annotated with a JML contract. This contract is sufficient from the implementation perspective, where our goal is to show absence of overflow. Before modification of the `item` field, the old item is temporarily stored and later returned to ensure the postcondition of the contract holds (see lines 10, 21 and 23). Although a `Node` is modified, the state of `nodeList` stays unaffected (line 9), as `nodeList` only refers to the nodes themselves in a particular order and has the same reference value even after updates to the `item` field of each node.

An example client of `LinkedList` that calls `set(int, Object)` is shown in Listing 11. What if (an incorrect) implementation of the `set()` method would secretly clear the `item` field of all other nodes? The contract as given in Listing 10 would still be valid, simply because its `ensures` clause, which specifies the postcondition of a method call, does not say anything about the item fields of those nodes. Further, type `Node` class is not publicly visible, since it is an internal construct that is only needed to link nodes to each other. So, from a client perspective, we cannot reason about the `item` fields of the nodes. This not only applies to the `set()` method, but also to other methods that change the state of an instance of `LinkedList` as well. We might say that from a client perspective all state changing methods are *underspecified* if `itemList` is omitted.

Listing 13 shows a contract of the `set()` method where items *are* taken into account. This contract is now specified without refering to `nodeList` at all, and it specifies the contents of the linked list. Thus, this contract is suitable from the client perspective. For example, to verify the client code in Listing 11, we could now infer that the `itemList` for all other indices than the supplied one remains the same, and thus the boolean condition returned must be true if `n` and `m` are different in-bound indices.

```
/*@
  @ also
  @ ...
  @ public normal_behavior
  @   requires
  @     o != null;
  @   ensures
  @     \result >= -1 && \result < nodeList.length;
  @   ensures
  @     \result == -1 ==>
  @       (\forall \bigint i; 0 <= i < nodeList.length;
  @         !o.equals(((Node)nodeList[i]).item));
  @   ensures
  @     \result >= 0 ==>
  @       (\forall \bigint i; \result < i <
  ↪   nodeList.length;
  @         !o.equals(((Node)nodeList[i]).item)) &&
  @       o.equals(((Node)nodeList[\result]).item);
  @*/
public /*@ strictly_pure @*/ int
lastIndexOf(/*@ nullable @*/ Object o) {
    int index = size;
    if (o == null) {
        ...
    } else {
        /*@
          @ maintaining
          @   (\forall \bigint i; index <= i <
          ↪   nodeList.length;
          @         !o.equals(((Node)nodeList[i]).item));
          @ maintaining
          @   0 <= index && index <= nodeList.length;
          @ maintaining
          @   0 < index && index <= nodeList.length ==>
          @     x == (Node)nodeList[index - 1];
          @ maintaining
          @   index == 0 <==> x == null;
          @ decreasing
          @   index;
          @ assignable
          @   \strictly_nothing;
          @*/
        for (Node x = last; x != null; x = x.prev) {
            index--;
            if (o.equals(x.item))
                return index;
        }
    }
    return -1;
}
```

Listing 12: Method `lastIndexOf(Object)` annotated with JML. Searches the list from last to first for an element. Returns −1 if this element is not present in the list; otherwise returns the index of the node that was equal to the argument. Only the contract and branch in which the argument is non-`null` is shown due to space restrictions. Methods such as `indexOf`, `removeFirstOccurrence` and `removeLastOccurrence` are very similar.

All the specifications we have produced are present in the on-line artifact [6]. The specifications, as they were part of the original publication [7], that focus on the implementation-side perspective are stored in the folder `implementation`. To enable client-side reasoning, we updated the contracts for several methods and their proofs using `itemList`. These can be found in the folder `clientfriendly`.

```
/*@
  @ also
  @ public normal_behavior
  @   requires
  @     0 <= index < itemList.length;
  @   ensures
  @     itemList[index] == element &&
  @     (\forall \bigint i; 0 <= i <
  ↪   itemList.length && i != index;
  @      itemList[i] == \old(itemList[i])) &&
  @     \result == \old(itemList[index]);
  @ public exceptional_behavior
  @   ...
  @*/
```

Listing 13: Item-based JML contract of `set(int, Object)`.

4.2 Verification

We now discuss the verification of the method contracts as they are specified above. We start by describing the general strategy we have used to verify proof obligations. We also describe in more detail how to produce a single proof, and for that we consider the method `lastIndexOf(Object)`. This gives a general feel how proving in KeY works. This method is neither trivial, nor very complicated to verify. In a similar way, we have produced proofs for each method contract that we have specified.

Overview of verification steps. When verifying a method, we first instruct KeY to perform symbolic execution. Symbolic execution is implemented by numerous proof rules that, roughly speaking, reduce the size of the program fragments in the modal operators in JavaDL. During symbolic execution, the goal sequent is automatically simplified, potentially giving rise to multiple branches. Since our class invariant contains a disjunction (either the list is empty or not), we must take care not to split early in the symbolic execution: this would lead to a duplication of work in the proof effort. Thus we instruct KeY to delay unfolding the class invariant. When symbolic execution is finished, goals may still contain updated heap expressions that must be simplified further. After this has been done, one can compare the open goals to the method body and its annotations, and see whether the open goals in KeY look familiar and check whether they are true.

In the remaining part of the proof the user must find an appropriate mix between interactive and automatic steps. If a sequent is provable, there may be multiple ways to construct a closed proof tree. At (almost) every step the user has a choice between applying steps manually or automatically. It requires some experience in choosing which rules to apply manually: clever rule application decreases the size of the proof tree. Certain rules are never applied automatically, such as the cut rule. The cut rule splits a proof tree into two parts by introducing a lemma, but suitably chosen lemmata may significantly reduce the size of a proof and thus the effort required to produce it. For example, the acyclicity property can be introduced using cut.

Understanding the proof on an intuitive, conceptual level can be hard. This can be especially the case when the proof is done with little interactive steps while having a somewhat large proof tree. The challenge is to find a good balance between interactive and automatic steps, such that on the one hand there are not too many unnecessary manual steps while on the other hand the proof still remains understandable.

Instead of introducing lemmas by using the cut rule, KeY also allows to add lemmas through the use of so-called *taclets* [13,14]. Taclets are a formalism in KeY for extending its built-in logic by declaring additional sorts, functions, axioms and proof rules. KeY also allows users to create their own user-defined taclets. Let us consider the acyclicity property as an example. This property is expressed in JML as follows:

```
(\forall \bigint i; 0 <= i < nodeList.length - 1;
    (\forall \bigint j; i < j < nodeList.length;
        nodeList[i] != nodeList[j]))
```

But we can also formulate is as a taclet, as follows:

```
\rules {
nodeList_equal{
\schemaVar \variable int i;
\schemaVar \variable int j;
\schemaVar \term Heap h;
\schemaVar \term Field nodeList;
\add( \forall i; \forall j;(0 <= i& i < j & j <
↪   seqLen(Seq::select(heap, self,nodeList)) ->
↪   ! any::seqGet(Seq::select(heap, self, nodeList),j)=
any::seqGet(Seq::select(heap, self, nodeList),i)) ==> )
};
}
```

The formalization of lemmas as taclets has the advantage over proof cuts that every time one needs the lemma, one can apply the taclet without having to repeat its proof. This is particularly useful for properties, such as acyclicity, that are used in multiple methods. By applying the cut rule, at each point in each method where a cut is done, the user has to reprove the property, e.g. that the acyclicity property is implied by the class invariant. With taclets, the proof is done only once and the lemma can then be used wherever the user wants. However, it is then still necessary to ensure the taclet is sound. In general KeY does not provide a mechanism that guarantees the soundness of user-defined taclets. Thus, their use risks the consistency of the system and the user has to be careful. Although KeY does supports the verification of soundness for a limited subset of taclets [15].

Verification example. The method `lastIndexOf` has two contracts: one involves a `null` argument, and another involves a non-`null` argument. Both proofs are similar. Moreover, the proof for `indexOf(...)` is similar but involves the `next` reference instead of the `prev` reference. This contract is interesting, since proving its correctness shows the absence of the overflow of the `index` variable.

Proposition `lastIndexOf(Object)` *as specified in Listing 12 is correct.*

Proof We describe how to proof can be constructed in KeY. Set strategy to default strategy, and set max. rules to 5,000 with the setting to delay expanding the class axiom. Finish symbolic execution on the main goal. Set strategy to 1,000 rules and select DefOps arithmetical rules. Automatically close all provable goals under the root. One goal remains. Perform update simplification macro on the whole sequent, perform propositional with split macro on the sequent, and close provable goals on the root of the proof tree. There is a remaining open goal:

– $\mathit{index} - 1 = 0 \leftrightarrow x.\texttt{prev} = \texttt{null}$: split the equivalence. First case (left implies right): suppose $\mathit{index} - 1 = 0$, then $x = \mathit{self}.\texttt{nodeList}[0] = \mathit{self}.\texttt{first}$ and $\mathit{self}.\texttt{first}.\texttt{prev} = \texttt{null}$: solvable through unfolding the invariant and equational rewriting. Now, second case, suppose $x.\texttt{prev} = \texttt{null}$. Then, either $\mathit{index} = 1$ or $\mathit{index} > 1$ (from splitting $\mathit{index} \geq 1$). The first of which is trivial (close provable goal), and the second one requires instantiating quantified statements from the invariant, leading to a contradiction. Since we have supposed $x.\texttt{prev} = \texttt{null}$, but $x = \mathit{self}.\texttt{nodeList}[\mathit{index} - 1]$ and $\mathit{self}.\texttt{nodeList}[\mathit{index} - 1].\mathit{prev} = \mathit{self}.\texttt{nodeList}[\mathit{index} - 2]$ and $\mathit{self}.\texttt{nodeList}[\mathit{index} - 2] \neq \texttt{null}$.

□

Interesting verification conditions. The acyclicity property is used to close verification conditions that arise as a result of potential aliasing of node instances: it is used as a separation lemma. For example, after a method that changed the `next` field of an existing node, we want to reestablish that all nodes remain reachable from the `first` through `next` fields (i.e., "connectedness"): one proves that the update of `next` only affects a single node, and does not introduce a cycle. We prove this by using the fact that two nodes instances are different if they have a different index in `nodeList`, which follows from acyclicity. Below, we sketch an argument why the acyclicity property follows from the invariant. We have a video in which we show how the argument in KeY goes, see [16, 0:55–11:30].

Proposition *Acyclicity follows from the linked list invariant.*

Proof By contradiction: suppose a linked list of size $n > 1$ is not acyclic. Then there are two indices, $0 \leq i < j < n$, such that the nodes at index i and j are equal. Then it must hold that for all $j \leq k < n$, the node at k is equal to the node at $k - (j - i)$. This follows from induction. Base case: if $k = j$, then node j and node $j - (j - i) = i$ are equal by assumption. Induction step: suppose node at k is equal to node at $k-(j-i)$, then if $k+1<n$ it also holds that node $k+1$ equals node $k+1-(j-i)$: this follows from the fact that node $k+1$ and $k+1-(j-i)$ are both the `next` of node $k<n-1$ and node $k-(j-i)$. Since the latter are equal, the former must be equal too. Now, for all $j \leq k < n$, node k equals node $k-(j-i)$ in particular holds when $k=n-1$. However, by the property that only the last node has a `null` value for `next`, and a non-last node has a non-`null` value for its `next` field, we derive a contradiction: if nodes k and $k-(j-i)$ are equal then all their fields must also have equal values, but node k has a `null` and node $k-(j-i)$ has a non-`null` next field! □

Summary of verification effort The total effort of our case study was about 7 man months. The largest part of this effort is finding correct and sufficiently complete specifications. KeY supports various ways to specify Java code: pure observer methods, model fields, and ghost fields. For example, using pure observer methods, contracts are specified by expressing the content of the list before/after a method by using the pure method `get(i)`, which returns the item at index i. This led to rather complex proofs: essentially it led to reasoning in terms of relational properties on programs (i.e. `get(i)` before vs `get(i)` after the method under consideration). After 2.5 man months of writing partial specifications and partial proofs in these different formalisms, we decided to go with ghost fields as this was the only formalism in which we succeeded to prove non-trivial methods.

It then took ≈ 4 man months of iterating in our workflow through (failed) partial proof attempts and refining the specs until they were sufficiently complete. In particular, changes to the class invariant were "costly", as this typically caused proofs of all the methods to break (one must prove that all methods preserve the class invariant). The possibility to interact with the prover was crucial to pinpoint the cause of a failed verification attempt, and we used this feature of KeY extensively to find the right changes/additions to the specifications.

In general there seems to be a hierarchy of specification levels: ghost fields, class invariants, method contracts, and loop invariants. A modification of a specification higher up in the hierarchy typically requires revisiting the specifications lower in the hierarchy. For example, without proper ghost fields it is difficult to express a useful class invariant. Method contracts implicitly depend on the class invariant. For a couple of methods, loop invariants had to be added as annotation. We almost always needed some iterations before finding a loop invariant that was sufficient. Moreover, a change in the method contract often implies its nested loop invariant need to be changed as well, to maintain the properties needed to show the contract is valid.

After the introduction of the field `nodeList`, several methods could be proved very easily, with a very low number of interactive steps or even automatically. Methods `unlink(Node)` and `linkBefore(Object, Node)` could not be proven without knowing the position of the node argument. We introduced a new ghost field, `nodeIndex`, that acts like a ghost parameter. Luckily, this did not affect the class invariant, and existing proofs that did not make use of the new ghost field were unaffected.

Once the specifications were (sufficiently) complete, we estimate that it only took approximately 1 or 1.5 man weeks to prove all methods. This can be reduced further if informal proof descriptions are given that describe the overall approach or outline of a proof. Moreover, we have recorded a video of a 30 minute proof session where the method `unlinkLast` is proven correct with respect to its contract [16]. Finally, we introduced client-friendly specifications and simplified the verification effort by introducing a taclet. We then reverified the most difficult methods only with respect to the new contracts and by applying taclets.

Proof statistics. As mentioned in Subsection 4.1, we first proved contracts that focus on correctness of the linked list from the implementation perspective. In our artifact [6] those can be found in the `implementation` folder. These proofs were constructed with KeY version 2.6.3. Subsequently, we added for several methods contracts that allow for client-side reasoning, using the previously mentioned `itemList`. By this time, KeY 2.8.0 was available so the proofs of those contracts (in folder `clientfriendly` of our artifact) were constructed with this new version. In the tables with the proof statistics below, we will clearly mark this distinction between versions. Table 1 summarizes the main proof statistics for all methods with contracts from the implementation perspective.

We found the most difficult proofs were for the method contracts of: `clear()`, `linkBefore(Object,Node)`, and `unlink(Node)`. Table 2 shows the proof statistics. The number of interactive steps seem a rough measure for effort required. But, we note that it is not a reliable representation of the difficulty of a proof: an experienced user can produce a proof with very few interactive steps, while an inexperienced user may take many more steps. The proofs we have produced are by no means minimal.

Initially, our main goal was to prove as many methods correct as possible, without striving to have proofs with a low number of interactive steps. After that goal was accomplished, we have recreated proofs for some methods with the goal to minimize the number of interactive steps. Table 3 shows the results for three methods. A first sight, surprisingly, the client-friendly contracts required less interactive steps (last column), while the contracts that were proven were stronger as they included the item list. The main reasons are that those proofs used the acyclicity taclet rather than proof cuts (so they do not include the proof of the acyclicity prop-

Table 1 Summary of proof statistics. Rules is the number of rules applied during the verification process, Branches is the number of branches in the proof tree, I.step is the number of interactive proof steps, Q.inst is the number of quantifier instantiation rules, O.Contract is the number of method contracts applied, and Loop inv. is the number of loop invariants. The last two columns are not metrics of the proof, but they indicate the total lines of code (LoC) and the total lines of specifications (LoS)

Rules	Branches	I.step	Q.inst	O.Contract	Loop inv.	LoC	LoS
375,839	2477	9609	2322	79	12	440	756

Table 2 Proof statistics of method contracts `clear`, `linkBefore` and `unlinkFirst`. The rows that are marked with † are of the client-friendly contracts. The other rows show the statistics for the proofs of the implementation-oriented contracts

Method	Rules	Branches	I.step	Q.inst	O.Contract	Loop inv.	LoC	LoS
clear	16,340	130	1068	35	0	1	12	42
clear†	11,921	73	92	26	0	1	12	48
linkBefore	28,146	222	531	109	0	0	10	13
linkBefore†	50,661	338	233	131	0	0	10	22
unlinkFirst	11,186	114	276	30	0	0	13	7
unlinkFirst†	15,611	108	70	50	0	0	13	16

Table 3 Total and interactive steps for methods `lastIndexOf()`,`linkFirst()` and `addFirst()` for the original proofs (first column), re-proofs (second column) and client-side friendly contracts (third column)

Method	Rules/I.step	Rules/I.step (reproof)	Rules/I.step (taclets†)
lastIndexOf(Object)	4571/23	4880/18	5194/9
linkFirst(Object)	33,650/665	13,974/24	10,933/22
addFirst(Object)	1863/31	612/0	3929/0

erty itself), and that the involved person doing the proofs was different and had more experience with KeY.

We now discuss the differences between two proof sessions as shown in the second column and third column of Table 3. For `lastIndexOf(Object)`, the difference in the number of interactive steps is negligible. This is since the proofs only differ in the choice of some specific low-level rules. For `linkFirst(Object)` the story is different: the number of interactive steps has decreased significantly (from 665 to 24). Unfolding the class invariant has been delayed until the terms it contains are really needed. Largely because of that, in the reproof the number of branches has decreased from 180 to 108 (not shown in the table). The number of quantifier instantiations went down from 277 to 47 (not shown in the table). The first two of the 24 interactive steps are both unpacking the class invariant; the first time as antecedent of the sequent, the second time as succedent of the sequent. The rest of the interactive steps is a mix of the following rules (each applied at least once):

– Splitting up the proof tree into two parts by applying
  - (A cleverly chosen instantiation of) a cut rule;
  - The impLeft rule;
  - The orLeft rule;
– Removing quantifiers (skolemization) by applying the allRight rule;
– Dividing a succedent partially as antecedent by applying the impRight rule;
– Applying substitutions by using the allLeft rule.

In addition, the following interactive steps have been applied: a rule that was needed for casting an expression (ineffectiveCast rule), a rule used for induction (auto_induction rule), and a rule to rewrite the $\leq$ operator (leq_to_gt_alt rule). The acyclicity property was needed and was introduced by applying a cut rule. The reason for the improvement of `addFirst(Object)` is the same as for `linkFirst (Object)`: delaying the unfolding of the class invariant. It is clear that this measure applies as a means to keep the number of interactive steps as low as possible. Not only for the methods mentioned here, but for the other methods as well. When the class invariant is unfolded too early, i.e., before the containing terms are actually needed, the proof tree too early grows too big unnecessarily. From that moment on it takes many steps, including interactive ones, to 'repair the damage', viz. closing all the unnecessary extra branches that are the result of the premature unfolding of the class invariant.

5 Discussion

In this section we discuss some of the main challenges of verifying the real-world Java implementation of a LinkedList, as opposed to the analysis of an idealized mathematical linked list. Many of these challenges are also relevant beyond our case study and apply to real-world library/program verification in general.

Challenge 1: Extensive use of Java language constructs The LinkedList class uses a wide range of Java language features. This includes nested classes (both static and non-static), inheritance, polymorphism, generics, exception handling, object creation and for each loops. To load and reason about the real-world LinkedList source code requires an analysis tool with high coverage of the Java language, including support for the aforementioned language features. However, since the Java language is a moving target and, more recently, new feature releases are published every 6 months (!), it is becoming more and more challenging for analysis tools to keep up [17]. For example, KeY already does not *fully* support generics, a relatively old feature that was added in Java 1.5. Even newer features, such as lambdas, functional interfaces and streams, typically build on top of older features such as generics.

Challenge 2: Support for intricate Java semantics The Java List interface is position based, and associates with each item in the list an index of Java's int type. The bugs described in Sect. 2.1 were triggered on large lists, in which integer overflows occurred. Thus, while an idealized mathematical integer semantics is much simpler for reasoning, it could not be used to analyze the bugs we encountered! It is therefore critical that the analysis tool faithfully supports Java's semantics, including Java's integer (overflow) behavior.

Challenge 3: Collections have a huge state space A Java collection is an object that contains other objects (of a reference type). Collections may grow to an arbitrary size (but in practice is bounded by available memory). By their very nature, collections thus intrinsically have a large state. To make this more concrete: triggering the bugs in LinkedList requires at least 2^{31} elements (and 64 GiB of memory), and each element, since it is of a reference type, has at least 2^{32} values.

Challenge 4: Interface specifications Several of the LinkedList methods contain an interface type as parameter. For example, the addAll method takes two arguments, the second one is of the Collection type:

```
public boolean addAll(int index, Collection c) {
        ...
        Object[] a = c.toArray();
        ...
}
```

As KeY follows the design by contract paradigm, verification of LinkedList's addAll method requires a contract for each of the other methods called, including the toArray method in the Collection *interface*. How can we specify interface methods, such as Collection.toArray? KeY features a stub generator that can automatically create conservative, default contracts: the method for which the contract is generated is specified to arbitrarily modify the heap and return *any* array. The default contract can then be made more precise manually. Simple conditions on parameters or the return value are easily expressed, but meaningful contracts that relates the behavior of the method to the contents of the collection require some notion of state to capture all mutations of the collection, so that previous calls to methods in the interface that contributed to the current contents of the collection are taken into account.

Model fields/methods [5, Section 9.2] are a widely used mechanism in KeY for abstract specification. A model field or method is represented in a concrete class in terms of the concrete state given by its fields. In this case, as only the interface type Collection is known rather than a concrete class, such a representation cannot be defined. Thus the behavior of the interface cannot be captured by specifications in terms of its state, which makes specification of methods such as Collection.toArray difficult. Ghost variables cannot be used either, since ghost variables are updated by adding set statements in method bodies, and interfaces do not contain method bodies. This raises the question: how to specify behavior of interface methods?[6]

Challenge 5: Verifiable code revisions We fixed the LinkedList class by explicitly bounding its maximum size to Integer.MAX_VALUE elements, but other solutions are possible. Rather than using integers indices for elements, one could change to an index of type long or BigInteger. Such a code revision is however incompatible with the general Collection and List interfaces (whose method signatures mandate the use of integer indices), thereby breaking all existing client code that uses LinkedList. Clearly this is not an option in a widely used language like Java, or any language that aims to be backwards compatible.

It raises the challenge: can we find code revisions that are compatible with existing interfaces and their clients? We can take this challenge even further: can we use our workflow to find such compatible code revisions, and are they also amenable to formal verification?

Challenge 6: Proof reuse Section 4.2 discussed the proof effort (in person months). It revealed that while the total effort was 6-7 person months, once the specifications are in place

[6] Since the representation of classes that implement the interface is unknown in the interface itself, a particularly challenging aspect here is: how to specify the footprint of an interface method, i.e.: what part of the heap can be modified by the method in the implementing class?

after many iterations of the workflow, producing the actual final proofs took only 1-2 weeks! But minor specification changes often require to redo nearly the whole proof, which causes much delay in finding the right specification. Other program verification case studies [2,5,18,19] show similarly that the main bottleneck today is specification, not verification. This calls for techniques to optimize proof reuse when the specification is slightly modified, allowing for a more rapid development of specifications.

Challenge 7: Code written without verification in mind Existing code in general is not designed to be easily suitable for verification. For example, the `LinkedList` class exposes several implementation details to classes in the java.util package: i.e., all fields, including `size`, are package private (not private!), which means they can be assigned a new value directly (without calling any methods) by other classes in that package. This includes setting `size` to negative values. As we have seen, the class malfunctions for negative `size` values. In short, this means that the `LinkedList` itself cannot enforce its own invariants anymore: its correctness now depends on the correctness of other classes in the package. The possibility to avoid calling methods to access the fields may yield a small performance gain, but it precludes a modular analysis: to assess the correctness of `LinkedList` one must now analyze all classes in the same package (!) to determine whether they make benign changes (if any) to the fields of the list. Hence, we recommend to encapsulate such implementation details, including making at least all fields `private`.

5.1 Status of the challenges

The KeY system covers the Java language features (Challenge 1) sufficiently to load and statically verify the `LinkedList` source code. KeY also supports various integer semantics, allowing us to analyze `LinkedList` with the actual Java integer overflow semantics (Challenge 2). As KeY is a theorem prover (based on deductive verification), it does not explore the state space of the class under consideration, thus solving the problem of the huge state space of Java collections (Challenge 3). We could not find any other analysis tools that solved these challenges, so we decided at that point to use KeY.

Concerning interface specifications (Challenge 4), progress has been made in [20,21]. This challenge can be addressed by specifying interfaces using an abstract state in terms of trace specifications, which consists of user-defined functions that map a sequence of calls to the interface methods to a value. This approach is very likely applicable in the context of `LinkedList`, however it does require rewriting the currently state-based specifications. It would be useful to investigate in future work whether a combination of trace-based and state-based specifications can be used so that existing state-based assertions can still be leveraged.

The challenges related to code revisions (Challenge 5) and proof reuse (Challenge 6) are compounded for analysis tools that use brittle proof representations. For example, proof files in KeY consist of actual rule applications (rather than higher level macro/strategy applications), and proof rule applications explicitly refer to the indices of the (sub)formulas the rule is applied to. This results in a fragile proof format, where small changes to the specifications or source code (such as a code refactoring) break the proof. Other state-of-the-art verification systems such as Coq, Isabelle and PVS support proof *scripts*. Those proofs are described at a higher level when compared to KeY. It would be interesting to see to what extent logics for verifying Java programs by means of symbolic execution can be handled in (extensions of) such higher-level proof script languages.

Finally, we challenge other existing static analysis tools to the (functional) verification of the Java Collection Framework, and techniques like model-checking for detecting the bugs we have found in the `LinkedList` implementation. We strongly believe that analysis of actual software as used in practice will provide a strong impetus to the further development of competing tools and techniques.

5.2 Related work

Knüppel et al. [12] provided a report on the specification and verification of some methods of the classes `ArrayList`, `Arrays`, and `Math` of the OpenJDK Collection Framework using KeY. Their report is mainly meant as a “stepping stone towards a case study for future research.” To the best of our knowledge, no formal specification and verification of the actual Java implementation of `LinkedList` as part of the Java Collection Framework has been investigated before: otherwise, the integer overflow bug would have been discovered before.

Gladisch and Tyszberowicz [22] have investigated the specification of linked data structures using JML, and focus on providing specifications in terms of pure observer methods instead of ghost fields. They define their own data structures, and do not specify and verify parts of the actual Java Collection Framework: the challenge in the latter is that it is not formally specified, nor specifically designed to be suitable for formal verification. One of their goals is to give specifications in JML that are suitable for both verification and testing. Ghost fields cannot be used for testing, as ghost fields are present only in comments at the source code level and are not affecting the execution. As we already mentioned in Sect. 4.2, verifying the linked list methods with respect to specifications given in terms of pure observer methods was too difficult for us and we decided to use ghost fields instead, which proved successful.

Zee et al. [23] have investigated full functional verification of several linked data structures in Java making use of specifications formulated in a classical higher-order logic. Also they define their own data structures, and do not specify and verify parts of the actual Java Collection Framework. Compared to our work, they are limited by their support of the Java language, e.g. lacking support for exceptions and they "currently model numbers as algebraic quantities with unbounded precision". Since our work involves showing the absence of integer overflow, it is crucial to have a realistic correspondence between the program semantics as employed during verification and the actual run-time behavior. Other formalizations of Java in higher-order logic also exists, such as Bali [24] and Jinja [25] (using the general-purpose theorem prover Isabelle/HOL), with similar limitations. The tools OpenJML [26] and VerCors [27] support verifying real-world Java programs. However, also these formalizations do not have a complete enough Java semantics to be able to analyze the bugs presented in this paper. In particular, the latter tool seems not to have built-in support for integer overflow arithmetic, although that can be added manually.

In general, the data structure of a linked list has been studied mainly in terms of pseudo code of an idealized mathematical abstraction (e.g., see [28] for an Eiffel version and [29] for a Dafny version). Lahiri and Qadeer [30] describe in general a technique for formalizing linked lists, and is related to our discussion of acyclicity. A comprehensive survey of the literature on specification and verification of linked lists is out of scope for this paper, but a good start are the related work sections within the above references.

Acknowledgements We thank the anonymous security engineers at Oracle for reviewing our bug report, and Bill Situ for handling our case. We furthermore thank the anonymous referees for their useful suggestions to improve this paper.

References

1. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's sort method for generic collections. J. Autom. Reason. **62**(1), 93–126 (2019). https://doi.org/10.1007/s10817-017-9426-4
2. de Gouw, S., Rot, J., de Boer, F.S., Bubel, R., Hähnle, R.: OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case. In: CAV 2015: Computer Aided Verification. LNCS, vol. 9206, pp. 273–289. Springer (2015)
3. Huisman, M., Ahrendt, W., Bruns, D., Hentschel, M.: Formal specification with JML. Tech. rep., Karlsruher Institut für Technologie (KIT) (2014). https://doi.org/10.5445/IR/1000041881
4. Leavens, G.T., Baker, A.L., Ruby, C.: JML: A notation for detailed design. In: Behavioral Specifications of Businesses and Systems, SECS, vol. 523, pp. 175–188. Springer (1999). https://doi.org/10.1007/978-1-4615-5229-1_12
5. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.) Deductive Software Verification: The KeY Book, LNCS, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
6. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY: Proof files (2021). https://doi.org/10.5281/zenodo.5648775
7. Hiep, H.A., Maathuis, O., Bian, J., de Boer, F.S., van Eekelen, M., de Gouw, S.: Verifying OpenJDK's LinkedList using KeY. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 12079, pp. 217–234. Springer (2020)
8. Klint, P., van der Storm, T., Vinju, J.: Rascal: A domain specific language for source code analysis and manipulation. In: 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation. pp. 168–177. IEEE (2009)
9. Ieu Eauvidoum, disk noise: Twenty years of escaping the Java sandbox. Phrack Magazine **0x10**(0x46), phile 0x07 of 0x0f (October 2021), http://www.phrack.org/issues/70/7.html#article
10. Camilo, F., Meneely, A., Nagappan, M.: Do bugs foreshadow vulnerabilities? a study of the Chromium project. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. pp. 269–279. IEEE (2015)
11. Knuth, D.E.: The Art of Computer Programming, vol. 1, 3rd edn. Addison-Wesley (1997)
12. Knüppel, A., Thüm, T., Pardylla, C., Schaefer, I.: Experience report on formally verifying parts of OpenJDK's API with KeY. In: F-IDE 2018: Formal Integrated Development Environment. EPTCS, vol. 284, pp. 53–70. OPA (2018). https://doi.org/10.4204/EPTCS.284.5
13. Giese, M.: Taclets and the KeY prover. Electron. Notes Theor. Comput. Sci. **103**, 67–79 (2004)
14. Habermalz, E.: Ein dynamisches automatisierbares interaktives Kalkül für schematische theorie spezifische Regeln. In: Ph.D. thesis, University of Karlsruhe (2000)
15. Bubel, R., Roth, A., Rümmer, P.: Ensuring the correctness of lightweight tactics for JavaCard dynamic logic. Electron. Notes Theor. Comput. Sci. **199**, 107–128 (2008)
16. Bian, J., Hiep, H.A.: Verifying OpenJDK's LinkedList Using KeY: Video (2019). https://doi.org/10.6084/m9.figshare.10033094.v2
17. Cok, D.R.: Jml and openjml for java 16. In: Proceedings of the 23rd ACM International Workshop on Formal Techniques for Java-like Programs. pp. 65–67 (2021)
18. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Lessons learned from microkernel verification—specification is the new bottleneck. In: SSV 2012: Systems Software Verification. EPTCS, vol. 102, pp. 18–32. OPA (2012). https://doi.org/10.4204/EPTCS.102.4
19. de Gouw, S., de Boer, F.S., Rot, J.: Proof Pearl: The KeY to correct and stable sorting. J. Autom. Reason. **53**(2), 129–139 (2014). https://doi.org/10.1007/s10817-013-9300-y
20. Hiep, H.A., Bian, J., de Boer, F.S., de Gouw, S.: History-based specification and verification of Java collections in KeY. In: Integrated Formal Methods (iFM). LNCS, vol. 12546, pp. 199–217. Springer (2020)

21. Bian, J., Hiep, H.A., de Boer, F.S., de Gouw, S.: Integrating ADTs in KeY and their application to history-based reasoning. In: Formal Methods (FM). LNCS, vol. 13047. Springer (2021), to appear
22. Gladisch, C., Tyszberowicz, S.: Specifying linked data structures in jml for combining formal verification and testing. Sci. Comput. Progr. **107-108**, 19–40 (2015). doi:https://doi.org/10.1016/j.scico.2015.02.005, https://www.sciencedirect.com/science/article/pii/S0167642315000398, selected Papers from the Brazilian Symposiums on Formal Methods (SBMF 2012 and 2013)
23. Zee, K., Kuncak, V., Rinard, M.: Full functional verification of linked data structures. SIGPLAN Not. **43**(6), 349–361 (2008). https://doi.org/10.1145/1379022.1375624
24. Nipkow, T., von Oheimb, D.: Java *light* is type-safe—definitely. In: POPL 1998: Principles of Programming Languages. pp. 161–170. ACM (1998). https://doi.org/10.1145/268946.268960
25. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM TOPLAS **28**(4), 619–695 (2006). https://doi.org/10.1145/1146809.1146811
26. Cok, D.R.: OpenJML: Software verification for Java 7 using JML, OpenJDK, and Eclipse. In: F-IDE 2014: Workshop on Formal Integrated Development Environment. EPTCS, vol. 149, pp. 79–92. OPA (2014). https://doi.org/10.4204/EPTCS.149.8
27. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: Verification of parallel and concurrent software. In: iFM 2017: Integrated Formal Methods. LNCS, vol. 10510, pp. 102–110. Springer (2017). doi:10.1007/978-3-319-66845-1_7
28. Polikarpova, N., Tschannen, J., Furia, C.A.: A fully verified container library. In: FM 2015: Formal Methods. LNCS, vol. 9109, pp. 414–434. Springer (2015). https://doi.org/10.1007/978-3-319-19249-9_26
29. Klebanov, V., Müller, P., et al.: The 1st verified software competition: Experience report. In: FM 2011: Formal Methods. LNCS, vol. 6664, pp. 154–168. Springer (2011). https://doi.org/10.1007/978-3-642-21437-0_14
30. Lahiri, S.K., Qadeer, S.: Verifying properties of well-founded linked lists. In: Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 115-126. POPL ’06, Association for Computing Machinery, New York, NY, USA (2006). ISBN: 1595930272, https://doi.org/10.1145/1111037.1111048