OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case*

Stijn de Gouw[1,2], Jurriaan Rot[3,1], Frank S. de Boer[1,3], Richard Bubel[4], and Reiner Hähnle[4]

[1] CWI, Amsterdam, The Netherlands
[2] SDL, Amsterdam, The Netherlands
[3] Leiden University, The Netherlands
[4] Technische Universität Darmstadt, Germany

Abstract. We investigate the correctness of TimSort, which is the main sorting algorithm provided by the Java standard library. The goal is functional verification with mechanical proofs. During our verification attempt we discovered a bug which causes the implementation to crash. We characterize the conditions under which the bug occurs, and from this we derive a bug-free version that does not compromise the performance. We formally specify the new version and mechanically verify the absence of this bug with KeY, a state-of-the-art verification tool for Java.

1 Introduction

Some of the arguments often invoked against the usage of formal software verification include the following: it is expensive, it is not worthwhile (compared to its cost), it is less effective than bug finding (e.g., by testing, static analysis, or model checking), it does not work for "real" software. In this article we evaluate these arguments in terms of a case study in formal verification.

The goal of this paper is functional verification of sorting algorithms written in Java with mechanical proofs. Because of the complexity of the code under verification, it is essential to break down the problem into subtasks of manageable size. This is achieved with *contract-based deductive verification* [3], where the functionality and the side effects of each method are precisely specified with expressive first-order contracts. In addition, each class is equipped with an invariant that has to be re-established by each method upon termination. These formal specifications are expressed in the Java Modeling Language (JML) [11].

We use the state-of-art Java verification tool KeY [4], a semi-automatic, interactive theorem prover, which covers nearly full sequential Java. KeY typically finds more than 99% of the proof steps automatically (see Sect. 6), while the remaining ones are interactively done by a human expert. This is facilitated by the use in KeY of symbolic execution plus invariant reasoning as its proof paradigm. That results in a close correspondence between proof nodes and symbolic program states which brings the experience of program verification somewhat close to that of debugging.

* Partly funded by the EU project FP7-610582 ENVISAGE and the NWO project 612.063.920 CoRE.

The work presented here was motivated by our recent success to verify executable Java versions of counting sort and radix sort in KeY with manageable effort [6]. As a further challenge we planned to verify a complicated sorting algorithm taken from the widely used OpenJDK core library. It turns out that *the default implementation* of Java's `java.util.Arrays.sort()` and `java.util.Collection.sort()` method is an ideal candidate: it is based on a complex combination of merge sort and insertion sort [12, 15]. It had a bug history[5], but was reported as fixed as of Java version 8. We decided to verify the implementation, stripped of generics, but otherwise completely unchanged and fully executable. The implementation is described in Sect. 2.

During our verification attempt we discovered that the fix to the bug mentioned above is in fact not working. We succeeded to characterize the conditions under which the bug occurs and results in a crash (Sect. 4) and from this we could derive a bug-free version (Sect. 5) that does not compromise the performance.

We provide a detailed experience report (Sect. 6) on the formal specification and mechanical verification of correctness and termination of the fixed version with KeY (Sects. 5, 6). Summarizing, our real-life case study shows that formal specification and verification, at least of library code, pays off, but also shows the limitations of current verification technology. In Sect. 7 we draw conclusions.

Related work. Several industrial case studies have already been carried out in KeY [13, 14, 1]. The implementation considered here and its proof is the most complex and one of the largest so far. The first correctness proof of a sorting algorithm is due to Foley and Hoare, who formally verified Quicksort by hand [9]. Since then, the development and application of (semi)-automated theorem provers has become standard in verification. The major sorting algorithms Insertion sort, Heapsort and Quicksort were proven correct by Filliâtre and Magaud [8] using Coq, and Sternagel [16] formalized a proof of Mergesort within the interactive theorem prover Isabelle/HOL.

Acknowledgment. We thank Peter Wong for suggesting to verify TimSort.

2 Implementation of TimSort

The default implementation of `java.util.Arrays.sort` for non-primitive types is TimSort, a hybrid sorting algorithm based on mergesort and insertion sort. The algorithm reorders a specified segment of the input array incrementally from left to right by finding consecutive (disjoint) sorted segments. If these segments are not large enough, they are extended using insertion sort. The starting positions and the lengths of the generated segments are stored on a stack. During execution some of these segments are merged according to a condition on the top elements of the stack, ensuring that the lengths of the generated segments are decreasing and the length of each generated segment is greater than the sum of the next two. In the end, all segments are merged, yielding a sorted array.

We explain the algorithm in detail based on the important parts of the Java implementation. The stack of runs (a sorted segment is called here a "run") is

[5] `http://bugs.java.com/view_bug.do?bug_id=8011944`

encapsulated by the object variable `ts`. The stack of starting positions and run lengths is represented by the arrays of integers `runBase` and `runLen`, respectively. The length of this stack is denoted by the instance variable `stackSize`. The main loop is as follows (with original comments):

Listing 1. Main loop of TimSort

```
do {
  // Identify next run
  int runLen = countRunAndMakeAscending(a, lo, hi, c);
  // If run is short, extend to min(minRun, nRemaining)
  if (runLen < minRun) {
      int force = nRemaining <= minRun ? nRemaining : minRun;
      binarySort(a, lo, lo + force, lo + runLen, c);
      runLen = force;
  }
  // Push run onto pending-run stack, and maybe merge
  ts.pushRun(lo, runLen);
  ts.mergeCollapse();
  // Advance to find next run
  lo += runLen;
  nRemaining -= runLen;
} while (nRemaining != 0);
// Merge all remaining runs to complete sort
assert lo == hi;
ts.mergeForceCollapse();
assert ts.stackSize == 1;
```

In each iteration of the above loop, the next run is constructed. First, a maximal ordered segment from the current position `lo` is constructed (the parameter `hi` denotes the upper bound of the entire segment of the array `a` to be sorted). This construction consists in constructing a maximal descending or ascending segment and reversing the order in case of a descending one. If the constructed run is too short (that is, less than `minRun`) then it is extended to a run of length `minRun` using binary insertion sort (`nRemaining` is the number of elements yet to be processed). Next, the starting position and the length of the run is pushed onto the stack of the object variable `ts` by the method `pushRun` below.

Listing 2. pushRun

```
private void pushRun(int runBase, int runLen) {
  this.runBase[stackSize] = runBase;
  this.runLen[stackSize] = runLen;
  stackSize++; }
```

The method `mergeCollapse` subsequently checks whether the invariant (lines 5—6 of Listing 3) on the stack of runs still holds, and merges runs until the invariant is restored (explained in detail below). When the main loop terminates, the method `mergeForceCollapse` completes sorting by merging all stacked runs.

Listing 3. mergeCollapse

```
1  /**
2   * Examines the stack of runs waiting to be merged and merges
3   * adjacent runs until the stack invariants are reestablished:
4   *     1. runLen[i - 3] > runLen[i - 2] + runLen[i - 1]
5   *     2. runLen[i - 2] > runLen[i - 1]
6   * This method is called each time a new run is pushed onto the stack,
7   * so the invariants are guaranteed to hold for i < stackSize upon
8   * entry to the method.
```

```
 9  */
10  private void mergeCollapse() {
11    while (stackSize > 1) {
12      int n = stackSize - 2;
13      if (n > 0 && runLen[n-1] <= runLen[n] + runLen[n+1]) {
14        if (runLen[n - 1] < runLen[n + 1])
15           n--;
16        mergeAt(n);
17      } else if (runLen[n] <= runLen[n + 1]) {
18        mergeAt(n);
19      } else {
20        break; // Invariant is established
21      }
22    }
23  }
```

The method `mergeCollapse` ensures that the top three elements of the stack satisfy the invariant given in the comments above. In more detail, let `runLen[n-1]` $= C$, `runlen[n]` $= D$, and `runLen[n+1]` $= E$ be the top three elements. Operationally, the loop is based on the following cases: 1. If $C \leq D + E$ and $C < E$ then the runs at `n-1` and `n` are merged. 2. If $C \leq D + E$ and $C \geq E$ then the runs at `n` and `n+1` are merged. 3. If $C > D + E$ and $D \leq E$ then the runs at `n` and `n+1` are merged. 4. If $C > D + E$ and $D > E$ then the loop exits.

3 Breaking the Invariant

We next show that the method `mergeCollapse` does not preserve the invariant in the entire run stack, contrary to what is suggested in the comments. To see this, consider as an example the situation where `runLen` consists of 120, 80, 25, 20, 30 on entry of `mergeCollapse`, directly after 30 has been added by `pushRun`. In the first iteration of the `mergeCollapse` loop there will be a merge at 25, since $25 \leq 20 + 30$ and $25 < 30$, resulting in (Listing 3, lines 15 and 16): $120^{\times}$, 80, 45, 30. In the second iteration, it is checked that the invariant is satisfied at 80 and 45 (lines 13 and 17), which is the case since $80 > 45 + 30$ and $45 > 30$, and `mergeCollapse` terminates. But notice that the invariant does not hold at 120, since $120 \leq 80 + 45$. Thus, `mergeCollapse` has not fully restored the invariant.

More generally, an error (violation of the invariant) can only be introduced by merging the second-to-last element and requires precisely four elements after the position of the error, i.e., at `runLen[stackSize-5]`. Indeed, suppose `runLen` consists of four elements A, B, C, D satisfying the invariant (so $A > B + C$, $B > C + D$ and $C > D$). We add a fifth element E to `runLen` using `pushRun`, after which `mergeCollapse` is called. The only possible situation in which an error can be introduced, is when $C \leq D + E$ and $C < E$. In this case, C and D will be merged, yielding the stack $A, B, C + D, E$. Then `mergeCollapse` checks whether the invariant is satisfied by the new three top elements. But A is not among those, so it is not checked whether $A > B + C + D$. As shown by the above example, this latter inequality does not hold in general.

3.1 The Length of `runLen`

The invariant affects the maximal size of the stack of run lengths during exection; recall that this stack is implemented by `runLen` and `stackSize`. The length of `runLen` is declared in the constructor of TimSort, based on the length of the input array `a` and, as shown below, on the assumption that the invariant holds. For performance reasons it is crucial to choose `runLen.length` as small as possible (but so that `stackSize` does not exceed it). The original Java implementation is as follows[6] (in a recent update the number 19 was changed to 24, see Sect. 4):

Listing 4. Bound of runLen based on length of the input array

```
int len = a.length;
int stackLen = (len <     120  ? 5 :
                len <    1542  ? 10 :
                len < 119151  ? 19 : 40);
```

We next explain these numbers, assuming the invariant to hold. Consider the sequence $(b_i)_{i\geq 0}$, defined inductively by $b_0 = 0$, $b_1 = 16$ and $b_{i+2} = b_{i+1} + b_i + 1$. The number 16 is a general lower bound on the run lengths. Now $b_0, \ldots, b_n$ are lower bounds on the run lengths in an array `runLen` of length n that satisfy the invariant; more precisely, $b_{i-1} \leq$ `runLen[n-i]` for all i with $0 < i \leq n$.

Let `runLen` be a run length array arising during execution, assume it satisfies the invariant, and let $n =$ `stackSize`. We claim that for any number B such that $1 + \sum_{i=0}^{B} b_i >$ `a.length` we have $n \leq B$ throughout execution. This means that B is a safe bound, since the number of stack entries never exceeds B.

The crucial property of the sequence (b_i) is that throughout execution we have $\sum_{i=0}^{n-1} b_i < \sum_{i=0}^{n-1}$ `runLen[i]` using that $b_0 = 0 <$ `runLen[n-1]` and $b_{i-1} \leq$ `runLen[n-i]`. Moreover, we have $\sum_{i=0}^{n-1}$ `runLen[i]` $\leq$ `a.length` since the runs in `runLen` are disjoint segments of `a`. Now for any B chosen as above, we have $\sum_{i=0}^{n-1} b_i < \sum_{i=0}^{n-1}$ `runLen[i]` $\leq$ `a.length` $< 1 + \sum_{i=0}^{B} b_i$ and thus $n \leq B$. Hence, we can safely take `runLen.length` to be the least B such that $1 + \sum_{i=0}^{B} b_i >$ `a.length`. If `a.length` < 120 we thus have 4 as the minimal choice of the bound, for `a.length` < 1542 it is 9, etc. This shows that the bounds used in OpenJDK (Listing 4) are slightly suboptimal (off by 1). The default value 40 (39 is safe) is based on the maximum $2^{31} - 1$ of integers in Java.

4 Worst Case Stack Size

In Sect. 3 we showed that the declared length of `runLen` is based on the invariant, but that the invariant is not fully preserved. However, this does not necessarily result in an actual error at runtime. The goal is to find a bad case, i.e., an input array for TimSort of a given length k, so that `stackSize` becomes larger than `runLen.length`, causing an ArrayIndexOutOfBoundsException in `pushRun`. In

[6] TimSort can also be used to sort only a segment of the input array; in this case, `len` should be based on the length of this segment. In the current implementation this is not the case, which negatively affects performance.

this section we show how to achieve the *worst case*: the maximal size of a stack of run lengths which does not satisfy the invariant. For certain choices of k this *does* result in an exception during execution of TimSort, as we show in Section 4.1. Not only does this expose the bug, our analysis also provides a safe choice for `runLen.length` that avoids the out-of-bounds exception.

The general idea is to construct a list of run lengths that leads to the worst case. This list is then turned into a concrete input array for TimSort by generating actual runs with those lengths. For instance, a list (2,3,4) of run lengths is turned into the input array (0,1,0,0,1,0,0,0,1) of length $k = 9$.

The sum of all runs should eventually sum to k. Hence, to maximize the stack size, the runs in the worst case are short. A run that breaks the invariant is too short, so the worst case occurs with a maximal number of runs that break the invariant. However, the invariant holds for at least half of the entries:

Lemma 1. *Throughout execution of TimSort, the invariant cannot be violated at two consecutive runs in* `runLen`*.*

Proof. Suppose, to the contrary, that two consecutive entries A and B of the run length stack violate the invariant. Consider the moment that the error at B is introduced, so A is already incorrect. The analysis of Sect. 3 reveals that there must be exactly four more entries after B on the stack (labelled $C \dots F$) satisfying $D \leq E + F$ and $D < F$ to trigger the merge below:

$$\begin{array}{llllll} A^{\times} & B & C & D & E & F \\ A^{\times} & B^{\times} & C & D+E & F & \end{array}$$

Merging stops here (otherwise $B^{\times}$ would be corrected), and we have 1. $D < F$ and 2. $C > D + E + F$. Next, consider the moment that C was generated. Since $A^{\times}$ is incorrect, C must be the result of merging some C_1 and C_2:

$$A \qquad B \qquad C_1 \qquad C_2 \qquad D'$$

This gives: 3. $C_1 + C_2 = C$, 4. $C_1 > C_2$, 5. $C_1 < D'$, 6. $D' \leq D$. Finally, all run lengths must be positive, thus: 7. $E > 0$. It is easy to check that constraints 1.–7. yield a contradiction. □

The above lemma implies that in the worst case, `runLen` has the form:

$$Y_n, X_n^{\times}, Y_{n-1}, X_{n-1}^{\times}, \dots, Y_1, X_1 \tag{1}$$

where each X_i invalidates the invariant, i.e., $X_i \leq Y_{i-1} + X_{i-1}$, and each Y_i satisfies it, i.e., $Y_i > X_i + Y_{i-1}$ (except when $i \leq 2$, since at least 5 elements are required to break the invariant). In the remainder of this section we show how to compute an input (in terms of run lengths) on which execution of TimSort results in a run length stack of the form (1).

Observe that the above sequence (1) can not be reached by simply choosing an input with these run lengths: each X_i would be merged away when X_{i-1} is pushed. Instead, we choose the input run lengths in such a way that each X_i arises as a sum of elements $x_1^i, \dots, x_{n_i}^i$ and each Y_i occurs literally in the input.

In order to calculate the X_i's, suppose the top three elements of the stack are X_i, Y_{i-1}, x_1^{i-1}. Since X_i must not be merged away, we have $X_i > Y_{i-1} + x_1^{i-1}$. Thus, the minimal choice of X_i's and Y_i's is:

$$X_i = Y_{i-1} + x_1^{i-1} + 1 \qquad Y_i = X_i + Y_{i-1} + 1 \tag{2}$$

The base cases are $X_1 = m$ (with $x_1^1 = m$) and $Y_1 = m + 4$, where $m = 16$ is the minimal run length. From (2) we then derive that $X_2 = 20 + 16 + 1 = 37$. The next step is to show how the elements x_j^i are computed from X_i, $i \geq 2$. To minimize the X_i's and Y_i's, each x_1^i should be as small as possible. Moreover, the merging pattern that arises while adding x_j^i's needs to preserve the previous X_{i+1} and Y_{i+1}, thus the top three elements of the stack before pushing x_j should be (omitting the index i from the x's for readability):

$$X_{i+1}, Y_i, x_1 + \ldots + x_{j-2}, x_{j-1}$$

Pushing x_j should then result in the merge:

$$X_{i+1}, Y_i, x_1 + \ldots + x_{j-2} + x_{j-1}, x_j\ .$$

and merging should stop, so $x_1 + \ldots + x_{j-1} > x_j$. The above merge only occurs when $x_1 + \ldots + x_{j-2} < x_j$. Thus, we obtain the desired merging behaviour by choosing the sequences $x_1, \ldots, x_{n_i}$ such that $X_i = x_1 + \ldots + x_{n_i}$ and

$$\text{for all } j \leq n_i: \quad x_j \geq m \text{ and } x_1 + \ldots + x_{j-2} < x_j < x_1 + \ldots + x_{j-1} \tag{3}$$

Further, x_1 should be chosen as small as possible to minimize X_{i+1} (2).

To compute such a sequence $x_1, \ldots, x_n$ from a number X, we distinguish between the case that X lies within certain intervals for which we have a fixed choice (with optimal x_1), and other ranges, for which we apply a default computation. The default computation starts with $x_n = X - (\lfloor \frac{X}{2} \rfloor + 1)$ and proceeds to compute $x_1, \ldots, x_{n-1}$ from $\lfloor \frac{X}{2} \rfloor + 1$. By repeatedly applying this computation, we always end up in one of the intervals for which we have a fixed choice. Because of space limitations, we treat only the fixed choices for the intervals $[m, 2m]$, $[2m+1, 3m+2]$ and $[3m+3, 4m+1]$. In the first case the only possible choice is $x_1 = X$. In the second case we take $x_1 = \lfloor \frac{X}{2} \rfloor + 1$ and $x_2 = X - x_1$. Finally, in the last case we take $x_1 = m + 1$, $x_2 = m$ and $x_3 = X - (x_2 + x_1)$.

Proposition 1. *For any X, the above strategy yields a sequence that satisfies* (3) *with a minimal value of x_1.*

Proof. We have fixed choices for any X in $[0, 2m]$, $[2m+1, 3m+2]$, $[3m+3, 4m+1]$, $[5m+5, 6m+5]$, $[8m+9, 10m+9]$, $[13m+15, 16m+17]$. An X in the first interval results in a sequence of length 1, in the second a sequence of length 2, etc. Except for the first two intervals $x_1 = m + 1$ is always chosen. The requirements (3) imply $x_1 > x_2 \geq m$, thus for any $X > m$, $x_1 = m + 1$ is the best we can hope for. Next, observe that if $x_1 = m + 1$ is produced for $X \in [l, r]$ then $x_1 = m + 1$, for any $X \in [2l - 1, 2r - 1]$ as well (since then $(\lfloor \frac{X}{2} \rfloor + 1) \in [l, r]$). Applying this to the interval $[3m+3, 4m+1]$ gives $[6m+5, 8m+1]$, which combined with $[5m+5, 6m+5]$ gives $[5m+5, 8m+1]$. We thus also get $[10m+9, 16m+1]$, and combining this with $[8m+9, 10m+9]$ yields $[8m+9, 16m+1]$. Combining the latter with $[13m+15, 16m+17]$ we obtain $[8m+9, 16m+17]$. Since this interval gives the optimal $x_1 = m+1$, so do $[16m+17, 32m+33]$, $[32m+33, 64m+65]$, etc. Hence, we have the minimal $x_1 = m+1$, for any $X \geq 8m+9$.

For $X \leq 8m+9$ a (tedious) case analysis shows minimality of x_1. $\square$

All in all, we have shown how to construct an input that generates the worst case which is of the form (1) and where each of the sequences of x^i_j's is constructed using the above strategy, yielding a minimal x^i_1 by Proposition 1.

Theorem 1. *An input corresponding to the sequence of run lengths as constructed above produces the largest possible stack of run lengths for a given input length, which does not satisfy the invariant.*

4.1 Breaking TimSort

We implemented the above construction of the worst case [7]. Executing TimSort on the generated input yields the following stack sizes (given array sizes):

array size	64	128	160	65536	131072	67108864	1073741824
required stack size	3	4	5	21	23	41	49
`runLen.length`	5	10	10	19 (24)	40	40	40

The table above lists the required stack size for the worst case of a given length. The third row shows the declared bounds in the TimSort implementation (see Listing 4). The number 19 was recently updated to 24 after a bug report[1].

This means that, for instance, the worst case of length 160 requires a stack size of 5, and thus the declared `runLen.length` = 10 suffices. Further observe that 19 does not suffice for arrays of length at least 65536, whereas 24 does. For the worst case of length 67108864, the declared bound 40 does not suffice, and running TimSort yields an unpleasant result:

Listing 5. Exception during exection of TimSort

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 40
        at java.util.TimSort.pushRun(TimSort.java:386)
        at java.util.TimSort.sort(TimSort.java:213)
        at java.util.Arrays.sort(Arrays.java:659)
        at TestTimSort.main(TestTimSort.java:18)
```

5 Verification of a fixed version

In Sect. 3 we showed that `mergeCollapse` does not fully re-establish the invariant, which led to an ArrayIndexOutOfBoundsException in `pushRun`. The previous section provides a possible workaround: adjust `runLen.length` using a worst-case analysis. That section also made clear that this analysis is based on an intricate argument that seems infeasible for a mechanized correctness proof.

Therefore, we provide a more principled solution. We fix `mergeCollapse` so that the class invariant *is* re-established, formally specify the new implementation in JML and provide a formal correctness proof, focussing on the most important specifications and proof obligations. This formal proof has been fully mechanized in the theorem prover KeY [4] (see Sect. 6 for an experience report).

Listing 6. Fixed version of mergeCollapse

```
 1  private void mergeCollapse() {
 2      while (stackSize > 1) {
 3          int n = stackSize − 2;
 4          if (  n >= 1 && runLen[n−1] <= runLen[n] + runLen[n+1]
 5              || n >= 2 && runLen[n−2] <= runLen[n] + runLen[n−1]) {
 6              if (runLen[n−1] < runLen[n+1])
 7                  n−−;
 8          } else if (runLen[n] > runLen[n+1]) {
 9              break; // Invariant is established
10          }
11          mergeAt(n);
12      }
13  }
```

Listing 6 shows the new version of `mergeCollapse`. The basic idea is to check validity of the invariant on the top 4 elements of `runLen` (lines 4, 5 and 8), instead of only the top 3, as in the original implementation. Merging continues until the top 4 elements satisfy the invariant, at which point we break out of the merging loop (line 9). We prove below that this ensures that *all* runs obey the invariant.

To obtain a human readable specification and a feasible (mechanized) proof, we introduce suitable abstractions using the following auxiliary predicates:

Name	Definition
elemBiggerThanNext2(arr, idx)	$(0 \leq idx \wedge idx + 2 < arr.length) \rightarrow$ $arr[idx] > arr[idx+1] + arr[idx+2]$
elemBiggerThanNext(arr, idx)	$0 \leq idx \wedge idx + 1 < arr.length \rightarrow$ $arr[idx] > arr[idx+1]$
elemLargerThanBound(arr, idx, v)	$0 \leq idx < arr.length \rightarrow arr[idx] \geq v$
elemInv(arr, idx, v)	elemBiggerThanNext2$(arr, idx) \wedge$ elemBiggerThanNext$(arr, idx) \wedge$ elemLargerThanBound(arr, idx, v)

Intuitively, the formula elemInv(`runLen`, i, 16) is that `runLen[i]` satisfies the invariant as given in lines 5—6 of Listing 3, and has length at least 16 (recall that this is a lower bound on the minimal run length). Aided by these predicates we are ready to express the formal specification, beginning with the class invariant.

Class Invariant. A class invariant is a property that all instances of a class should satisfy. In a design by contract setting, each method is proven in isolation (assuming the contracts of methods that it calls), and the class invariant can be assumed in the precondition and must be established in the postcondition, as well as at all call-sites to other methods. The latter ensures that it is safe to assume the class invariant in a method precondition. A precondition in JML is given by a `requires` clause, and a postcondition is given by `ensures`. To avoid manually adding the class invariant at all these points, JML offers an `invariant` keyword which *implicitly* conjoins the class invariant to all pre- and postconditions. A seemingly natural candidate for the class invariant states that all runs on the stack satisfy the invariant and have a length of at least 16. However, `pushRun` does not preserve this invariant. Further, inside the loop of `mergeCollapse` (Listing 6) the `mergeAt` method is called, so the class invariant must hold. But at that point the invariant can be temporarily broken by the last 4 runs in `runLen` due to ongoing merging. Finally, the last run pushed on the stack in the main sorting loop (Listing 1) can be shorter than 16 if fewer items remain. The class invariant given below addresses all this:

Listing 7. Class invariant of TimSort

```
 1  /*@ invariant
 2  @      runBase.length == runLen.length
 3  @ && (a.length < 120 ==> runLen.length==4)
 4  @ && (a.length >= 120 && a.length < 1542 ==> runLen.length==9)
 5  @ && (a.length>=1542 && a.length<119151 ==> runLen.length==18)
 6  @ && (a.length >= 119151 ==> runLen.length==39)
 7  @ && (runBase[0] + (\sum int i; 0<=i && i<stackSize; (\bigint)runLen[i]) <= a.length)
 8  @ && (0 <= stackSize && stackSize <= runLen.length)
 9  @ && (\forall int i; 0<=i && i<stackSize-4; elemInv(runLen, i, 16))
10  @ && elemBiggerThanNext(runLen, stackSize-4)
11  @ && elemLargerThanBound(runLen, stackSize-3, 16)
12  @ && elemLargerThanBound(runLen, stackSize-2, 16)
13  @ && elemLargerThanBound(runLen, stackSize-1, 1)
14  @ && elemLargerThanBound(runBase, 0, 0)
15  @ && (\forall int i; 0<=i && i<stackSize-1;
16  @       (\bigint)runBase[i] + runLen[i] == runBase[i+1]);
17  @*/
```

Lines 3–6 specify the length of `runLen` in terms of the length of the input array `a`. Line 7–8 formalizes the property that the length of all runs together (i.e., the sum of all run lengths) does not exceed `a.length`. Line 9 contains bounds for `stackSize`. Line 10 expresses that all but the last 4 elements satisfy the invariant. The properties satisfied by the last 4 elements are specified on lines 11–14. Lines 15–17 formalize that run i starts at `runBase[i]` and extends for `runLen[i]` elements. As JML by default uses Java integer types, which can overflow, we need to make sure this does not happen by casting those expressions that potentially can overflow to `\bigint`.

The pushRun method. This method adds a new run of length `runLen` to the stack, starting at index `runBase`[7]. Lines 4–5 of Listing 8 express that the starting index of the new run (`runBase`) directly follows after the end index of the last run (at index `stackSize-1` in `this.runLen` and `this.runBase`). The assignable clause indicates which locations can be modified; intuitively the assignable clause below says that previous runs on the stack are unchanged.

Listing 8. Contract of pushRun

```
1  /*@ normal_behavior
2  @ requires
3  @      runLen > 0 && runLen <= a.length && runBase >= 0
4  @ && (stackSize > 0  ==> runBase ==
5  @   (\bigint)this.runBase[stackSize-1]+this.runLen[stackSize-1])
```

[7] These parameters shadow the instance variables with the same name; to refer to the instance variables in specifications one prefixes `this`, just like in Java.

```
 6  @ && ((\bigint)runLen + runBase <= a.length)
 7  @ && (\forall int i; 0<=i && i<ts.stackSize−2; elemInv(ts.runLen,i,16))
 8  @ && elemBiggerThanNext(ts.runLen, ts.stackSize−2)
 9  @ && elemLargerThanBound(ts.runLen, ts.stackSize−1, 16)
10  @ ensures
11  @    this.runBase[\old(stackSize)] == runBase
12  @ && this.runLen[\old(stackSize)] == runLen
13  @ && stackSize == \old(stackSize)+1;
14  @ assignable
15  @   this.runBase[stackSize], this.runLen[stackSize], this.stackSize;
16  @*/
17  private void pushRun(int runBase, int runLen)
```

The mergeCollapse method. The new implementation of `mergeCollapse` restores the invariant at all elements in `runLen`; this is stated in lines 6–7 of Listing 9. As `mergeCollapse` just merges existing runs, the sum of all run lengths should be preserved (lines 8–9). Line 10 expresses that the length of the last run on the stack after merging never decreases (merging increases it). This is needed to ensure that all runs, except possibly the very last one, have length ≥ 16.

Listing 9. Contract of mergeCollapse

```
 1  /*@ normal_behavior
 2  @ requires
 3  @      stackSize > 0  &&  elemInv(runLen, stackSize−4, 16)
 4  @   && elemBiggerThanNext(runLen, stackSize−3);
 5  @ ensures
 6  @    (\forall int i; 0<=i && i<stackSize−2;  elemInv(runLen, i, 16))
 7  @ && elemBiggerThanNext(runLen, stackSize−2)
 8  @ && ((\sum int i; 0<=i && i<stackSize; (\bigint)runLen[i])
 9  @      == \old((\sum int i; 0<=i && i<stackSize; (\bigint)runLen[i])))
10  @ && (runLen[stackSize−1] >= \old(runLen[stackSize−1]))
11  @ && (0 < stackSize && stackSize <= \old(stackSize));
12  @*/
13  private void mergeCollapse()
```

The loop invariant of `mergeCollapse` is given in Listing 10. As discussed above, merging preserves the sum of all run lengths (lines 2–3). Line 4 expresses that all but the last four runs satisfy the invariant: a merge at index `stackSize-3` (*before* merging) can break the invariant of the run at index `stackSize-4` *after* merging (beware: `stackSize` was decreased). Lines 5–8 state the conditions satisfied by the last 4 runs. Lines 9–10 specify consistency between `runLen` and `runBase`. The last line states that `stackSize` can only decrease through merging.

Listing 10. Loop Invariant of mergeCollapse

```
 1  /*@ loop_invariant
 2  @           ((\sum int i; 0<=i && i<stackSize; runLen[i])
 3  @       == \old((\sum int i; 0<=i && i<stackSize; runLen[i])))
 4  @ && (\forall int i; 0<=i && i<stackSize−4;  elemInv(runLen, i, 16))
 5  @ && elemBiggerThanNext(runLen, stackSize−4)
 6  @ && elemLargerThanBound(runLen, stackSize−3, 16)
 7  @ && elemLargerThanBound(runLen, stackSize−2, 16)
 8  @ && elemLargerThanBound(runLen, stackSize−1, 1)
 9  @ && (\forall int i; 0<=i && i<stackSize−1;
10  @       (\bigint)runBase[i] + runLen[i] == runBase[i+1])
11  @ && (runLen[stackSize−1] >= \old(runLen[stackSize−1]))
12  @ && (0 < stackSize && stackSize <= \old(stackSize));
13  @*/
```

To prove the contracts, several verification conditions must be established. We discuss the two most important ones. The first states that on entry of `pushRun`, the `stackSize` must be smaller than the stack length. The ArrayIndex-OutOfBoundsException of Listing 5 was caused by the violation of that property:

`requires(pushRun) &&` cl. invariant `==> stackSize < this.runLen.length`

Proof. Line 9 of the class invariant implies `stackSize` $\leq$ `this.runLen.length`. We derive a contradiction from `stackSize` $=$ `this.runLen.length` by considering four cases: `a.length` < 120, or `a.length` ≥ 120 `&&` `a.length` < 1542, or `a.length` ≥ 1542 `&&` `a.length` < 119151, or `a.length` ≥ 119151. We detail the case `a.length` < 120, the other cases are analogous. Since `a.length` < 120, line 3 of the class invariant implies `stackSize` $=$ `this.runLen.length` $= 4$.

Let `SUM` $=$ `this.runLen[0]` $+ \ldots +$ `this.runLen[3]`. Suitable instances of lines 16–17 of the class invariant imply `this.runBase[3]` $+$ `this.runLen[3]` $=$ `this.runBase[0]` $+$ `SUM`. Together with line 15 of the class invariant and lines 4–6 of the `pushRun` contract we get `runLen` $+$ `SUM` < 120. But the `\requires` clause of `pushRun` implies `runLen` > 0, so `SUM` < 119. The `\requires` clause also implies `runLen[3]` ≥ 16 (line 9), `runLen[2]` ≥ 17 (line 8), `runLen[1]` ≥ 34 and `runLen[0]` ≥ 52 (line 7). So `SUM` $\geq 16+17+34+52 = 119$, a contradiction. □

The second verification condition arises from the break statement in the `mergeCollapse` loop (Listing 6, line 9). At that point the guards on line 4 and 5 are false, the one on line 8 is true, and the `\ensures` clause of `mergeCollapse` (which implies that the invariant holds for all runs in `runLen`) must be proven:

$$\begin{pmatrix} \text{loop invariant of } \texttt{mergeCollapse \&\& n} = \texttt{stackSize-2} \\ \quad \texttt{\&\& n} > 0 \texttt{ ==> runLen[n-1]} > \texttt{runLen[n]} + \texttt{runLen[n+1]} \\ \quad \texttt{\&\& n} > 1 \texttt{ ==> runLen[n-2]} > \texttt{runLen[n-1]} + \texttt{runLen[n]} \\ \quad \texttt{\&\& n} >= 0 \texttt{ ==> runLen[n]} > \texttt{runLen[n+1]} \end{pmatrix}$$
$$\texttt{==> \textbackslash ensures(mergeCollapse)}$$

Proof. Preservation of sums (lines 8–9 of `\ensures`) follows directly from lines 2–3 of the loop invariant. Lines 10–11 of `\ensures` are implied by lines 11–12 of the loop invariant. The property `elemBiggerThanNext(runLen, stackSize-2)` follows directly from `n >= 0 ==> runLen[n] > runLen[n+1]`. We show by cases that `\forall int i; 0<=i && i<stackSize-2; elemInv(runLen, i, 16)`.

- `i` $<$ `stackSize` $- 4$: from line 4 of the loop invariant.
- `i` $=$ `stackSize` $- 4$: from line 3 of the premise. The original `mergeCollapse` implementation (Listing 3) did not cover this case, which was the root cause that the invariant `elemInv(runLen, i, 16)` could be false for some `i`.
- `i` $=$ `stackSize` $- 3$: from the second line of the premise. □

Of course, these proof obligations (plus all others) were formally shown in KeY.

5.1 Experimental evaluation

The new version of `mergeCollapse` passes all relevant OpenJDK unit tests[8]. However, it introduces a potential extra check in the loop, which might affect

[8] `http://hg.openjdk.java.net/jdk7u/jdk7u/jdk/file/70e3553d9d6e/test/java/util/Arrays/Sorting.java`

performance. We compared the new version with the OpenJDK implementation using the benchmark created by the original author of the Java port of TimSort. This benchmark is part of OpenJDK[9]. It generates input of several different types, of varying sizes and repetitions. We executed the benchmark on three different setups: (Sys. 1): MacBookPro, Intel Core i7 @ 2.6 GHz, 8 GB, 4 core; (Sys. 2): Intel Core i7 @ 2.8 GHz, 6 GB, 2 core; (Sys. 3): Intel(R) Core(TM) i7 @ 3.4 GHz, 16GB, 4 core. The table below summarizes the average speedup over 25 runs on each setup (see [7] for full results). The speedup is computed by dividing the benchmark result of the new version by the result of the original version. Thus, a value larger than 1 means that the new version wins.

	Sys. 1	Sys. 2	Sys. 3	Average
ALL_EQUAL_INT	0.9796	1.0094	1.0058	0.9983
ASCENDING_10_RND_AT_END_INT	0.9982	0.9997	0.9942	0.9974
ASCENDING_3_RND_EXCH_INT	1.0084	1.0130	1.0021	1.0079
ASCENDING_INT	0.9810	1.0082	1.0039	0.9977
DESCENDING_INT	0.9740	0.9897	0.9868	0.9835
DUPS_GALORE_INT	0.9910	0.9980	0.9981	0.9957
PSEUDO_ASCENDING_STRING	0.9652	0.9926	0.9929	0.9836
RANDOM_BIGINT	1.0064	1.0057	1.0047	1.0056
RANDOM_INT	0.9912	0.9989	0.9993	0.9965
RANDOM_WITH_DUPS_INT	0.9956	0.9971	0.9999	0.9975
WORST_CASE	1.0062	1.0075	1.0127	1.0088
All together (average)	0.9906	1.0018	1.0000	0.9975

The first column contains the type of input. We added WORST_CASE, which generates the worst case as presented in Section 4. This case is important because it discriminates the two versions as much as possible. The other types of input are defined in `ArrayBuilder.java` which is part of the OpenJDK benchmark. We conclude that the new version does not negatively affect the performance.

6 Experience with KeY

We constructed a mechanized proof in KeY, showing correctness of the class invariant, the absence of exceptions and termination for all methods that affect the bug. Due to the complexity of Timsort, this requires interactivity as well as powerful automated search strategies.

However, two methods (`mergeLo` and `mergeHi`) we did not manage despite a considerable effort. Each has over 100 lines of code and exhibits a complex control flow with many nested loops, six breaks, and several `if`-statements. This leads to a memory overflow while proving due to an explosion in the number of symbolic execution paths. These methods obviously do not invalidate the class invariant as they do not access `runLen` and `runBase`. All other 15 methods were fully verified, which required specifications of all methods. In total, there are 460 lines of specifications, compared to 928 lines of code (including whitespace).

Our analysis resulted in one of the largest case studies carried out so far in KeY with over 2 million proof steps in total. The KeY proof targets the ac-

[9] `http://hg.openjdk.java.net/jdk7u/jdk7u/jdk/file/70e3553d9d6e/test/java/util/TimSort`

tual implementation in the OpenJDK standard library, rather than an idealized model of it. That implementation uses low-level bitwise operations, abrupt termination of loops and arithmetic overflows. This motivated several improvements to KeY, such as new support for reasoning about operations on bit-vectors.

	Rule Apps	Interact	Call	Loop	Q-inst	Spec	LoC
binarySort	536.774	470	3	2	16	27	35
sort(a,lo,hi,c)	235.632	695	14	1	54	38	52
mergeCollapse	415.133	1529	7	1	225	48	13
mergeAt	279.155	690	4	0	1064	32	39
pushRun	26.248	94	0	0	69	18	5
mergeForceColl	53.814	294	1	1	113	39	10
Other (sum)	664.507	1257	135	20	195	132	179
Total	2.211.263	5029	164	25	1736	334	333

One reason for the large number of proof steps is their fine granularity. However, notice that only a relatively small number was applied manually ("Interact"). Most of the manual interactions are applications of elementary weakening rules (hiding large irrelevant formulas) for guiding the automated proof search. Approximately 5-10% required ingenuity, such as introducing crucial lemmas and finding suitable quantifier instantiations ("Q-inst"). The columns ("Call") and ("Loop") show the number of rule applications concerning calls and loops encountered in symbolic execution paths. Since multiple paths can lead to the same call, this is higher than the number of calls in the source code. The last two columns show the number of lines of specification and code (without comments).

The specification was constructed incrementally, by repeated attempts to complete the proof and, when failing, enhancing the (partial) specifications based on the feedback given by KeY. In particular, KeY can provide a symbolic counter example. For instance, KeY produces the following uncloseable goal when verifying the original `mergeCollapse` implementation:

```
runLen[stackSize−3] > runLen[stackSize−2] + runLen[stackSize−1],
\forall int i; 0<=i && i<stackSize−4; runLen[i] > runLen[i+1]+runLen[i+2]
==> runLen[stackSize−4] > runLen[stackSize−3] + runLen[stackSize−2]
```

The quantified formula says: the invariant holds except for the last five runs. The first formula establishes it for the last three runs. Nevertheless, it is broken by the fourth-last run, as the right hand side states. This information shows precisely where the invariant breaks (Section 3) and suggests how to fix the algorithm (Section 5): add a test for index `stackSize`-4 "somewhere". Due to symbolic execution, KeY produces proof trees that correspond closely to the structure of the program. This allows identifying *where* to add the extra check in the code.

While specifications were written incrementally, small changes to the class invariant required reproving instance methods almost from scratch. Indeed, a major challenge for properly supporting this incremental process is: how to avoid proof duplication? This could be partially addressed by introducing user-defined predicates to abstract from certain concrete parts of the specification. KeY already supports ad hoc introduction of user-defined predicates (Section 5). A systematic treatment is given in [5, 10]; its integration in KeY is ongoing work.

To reduce the number of symbolic paths, we heavily used block contracts around if-statements as a form of state merging. Current work focusses on more general techniques for merging different symbolic execution branches.

7 Conclusion and Future Work

Beyond the correctness result obtained in this paper, our case study allows to draw a number of more general conclusions:

1. State-of-art formal verification systems allow to prove functional correctness of actual implementations of complex algorithms that satisfy a minimum degree of structure and modularity.
2. Even core library methods of mainstream programming languages contain subtle bugs that can go undetected for years. Extensive testing was not able to exhibit the bug. Sections 3 and 4 explain why: the smallest counterexample is an array of 67+ million elements (with non-primitive type) and a very complex structure. It is interesting to note that the affected sorting implementation was ported to Java from the Python library.[10] It turns out that the bug is present in Python as well, ever since the method was introduced.[11] It can be fixed in the same manner as described above. Though the bug is unlikely to occur by accident, it can be used in denial-of-service attacks[12].
3. Software verification is often considered too expensive. However, precise formal specification allowed us to discover that the invariant is not preserved, in an afternoon. Section 6 shows that this fact also inevitably arises during verification with KeY. The combination of interactivity with powerful automated strategies was essential to formally verify the fixed version.
4. Static analysis and model checking are not precise, expressive and modular enough to fully capture the functionality of the involved methods. Expressive contracts are crucial to break down the problem into feasible chunks.

We conclude that functional deductive verification of core libraries of mainstream programming languages with expressive, semi-automated verification tools is feasible. To reach beyond the current limits, improvements based on program transformations, refinement, and proof reuse are mandatory. Further, it is clearly worthwhile: the OpenJDK implementation of `sort()` is used daily in billions of program runs, often in safety- or security-critical scenarios. The infamous Intel Pentium bug cost a lot of revenue and reputation, even though the actual occurrence of a defect was not more likely than in the case of TimSort. Since then, formal verification of microprocessors is standard (e.g., [2]). Isn't it time that we begin to apply the same care to core software components?

[10] `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`

[11] As the Python version works with 64bit integer types and uses larger bounds for `runLen`, it is even more unlikely to occur, however.

[12] `http://bugs.java.com/view_bug.do?bug_id=6804124`

References

1. W. Ahrendt, W. Mostowski, and G. Paganelli. Real-time Java API specifications for high coverage test generation. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 145–154, New York, NY, USA, 2012. ACM.
2. B. Akbarpour, A. T. Abdel-Hamid, S. Tahar, and J. Harrison. Verifying a synthesized implementation of IEEE-754 floating-point exponential function using HOL. *Comput. J.*, 53(4):465–488, 2010.
3. B. Beckert and R. Hähnle. Reasoning and verification. *IEEE Intelligent Systems*, 29(1):20–29, Jan.–Feb. 2014.
4. B. Beckert, R. Hähnle, and P. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer-Verlag, 2006.
5. R. Bubel, R. Hähnle, and M. Pelevina. Fully abstract operation contracts. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation, 6th International Symposium, ISoLA 2014, Corfu, Greece*, LNCS. Springer, Oct. 2014. In this proceedings.
6. S. de Gouw, F. S. de Boer, and J. Rot. Proof pearl: The KeY to correct and stable sorting. *Journal of Automated Reasoning*, 53(2):129–139, 2014.
7. S. de Gouw et al. Web appendix of this paper, 2015. `http://envisage-project.eu/?page_id=1412`.
8. J.-C. Filliâtre and N. Magaud. Certification of sorting algorithms in the system Coq. In *Theorem Proving in Higher Order Logics: Emerging Trends*, Nice, France, 1999.
9. M. Foley and C. A. R. Hoare. Proof of a recursive program: Quicksort. *Computer Journal*, 14(4):391–395, 1971.
10. R. Hähnle, I. Schaefer, and R. Bubel. Reuse in software verification by abstract method calls. In M. P. Bonacina, editor, *Proc. 24th Conference on Automated Deduction (CADE), Lake Placid, USA*, volume 7898 of *LNCS*, pages 300–314. Springer-Verlag, 2013.
11. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and WernerDietl. *JML Reference Manual*, May 2013. Draft revision 2344.
12. P. M. McIlroy. Optimistic sorting and information theoretic complexity. In V. Ramachandran, editor, *Proc. Fourth Annual ACM/SIGACT-SIAM Symposium on Discrete Algorithms, Austin*, pages 467–474. ACM/SIAM, 1993.
13. W. Mostowski. Formalisation and verification of Java Card security properties in dynamic logic. In M. Cerioli, editor, *Proc. Fundamental Approaches to Software Engineering (FASE), Edinburgh*, volume 3442 of *LNCS*, pages 357–371. Springer-Verlag, Apr. 2005.
14. W. Mostowski. Fully verified Java Card API reference implementation. In B. Beckert, editor, *Proc. 4th Intl. Verification Workshop in connection with CADE-21, Bremen, Germany*, volume 259 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
15. T. Peters. Timsort description, accessed February 2015. `http://svn.python.org/projects/python/trunk/Objects/listsort.txt`.
16. C. Sternagel. Proof Pearl - A mechanized proof of GHC's mergesort. *J. Autom. Reasoning*, 51(4):357–370, 2013.