A Generic Approach to the Verification of the Permutation Property of Sequential and Parallel Swap-Based Sorting Algorithms

Mohsen Safari(✉) and Marieke Huisman

Formal Methods and Tools, University of Twente, Enschede, The Netherlands
{m.safari,m.huisman}@utwente.nl

Abstract. Sorting is one of the fundamental operations in computer science, and many sequential and parallel algorithms have been proposed in the literature. Swap-based sorting algorithms are one category of sorting algorithms where elements are swapped repeatedly to achieve the desired order. Since these algorithms are widely used in practice, their (functional) correctness, i.e., proving sortedness and permutation properties, is of utmost importance. However, proving the permutation property using automated program verifiers is much more challenging as the formal definition of this property involves existential quantifiers. In this paper, we propose a generic pattern to verify the permutation property for any sequential and parallel swap-based sorting algorithm automatically. To demonstrate our approach, we use VerCors, a verification tool based on separation logic for concurrent and parallel programs, to verify the permutation property of bubble sort, selection sort, insertion sort, parallel odd-even transposition sort, quick sort, two in-place merge sorts and TimSort for any arbitrary size of input.

Keywords: Sorting algorithms · Deductive verification · Separation logic

1 Introduction

Sorting is one of the fundamental and frequently used operations in computer science. Sorting algorithms take a list of elements as input and rearrange them into a particular order as output. Sorting has many applications in searching, data structures and data bases. Because of its importance, the literature contains many sorting algorithms with different complexity. One category of sorting algorithms is swap-based sorting, where the elements are swapped repeatedly until the desired order is achieved (e.g., bubble sort).

Because of the increase in the amount of data and emerging multi-core architectures, also parallel versions of sorting algorithms have been proposed. For

This work is supported by NWO grant 639.023.710 for the Mercedes project.

B. Dongol and E. Troubitsyna (Eds.): IFM 2020, LNCS 12546, pp. 257–275, 2020.
https://doi.org/10.1007/978-3-030-63461-2_14

instance, odd-even transposition sort [16] has been proposed as a parallel version of the bubble sort algorithm. Parallelizing algorithms on many-core processors (e.g., GPGPUs) is an active area of research, and it has been shown that parallel (GPU-based) implementations of sorting algorithms [14,15,18,19,21,23] outperform their sequential (CPU-based) counterparts.

Due to the frequent use of both sequential and parallel sorting algorithms, their correctness is of utmost importance, which means that they must have the following properties: (1) sortedness: the output is ordered, and (2) permutation: the output is a permutation of the input (i.e., the elements do not change).

To establish these two properties of sorting algorithms, one can use dynamic approaches (e.g., testing) and run the programs with concrete inputs to find bugs. However, this does not guarantee the absence of bugs. In contrast, with static verification, the complete state space of a program is analyzed without running it. In deductive verification, a program is annotated with intermediate (invariant) properties. Then, using a program logic, the annotated code is translated into proof obligations which are discharged by an automated theorem prover.

Using deductive program verification, proving the permutation property is harder than the sortedness property. This might be surprising at first glance, since in the swap-based sorting algorithms, the main operation is only swapping two elements repeatedly. But the permutation property typically requires reasoning about existential quantifiers, which is challenging for the underlying automated theorem provers.

As discussed by Filliatre [9], there are three common solutions: (1) in a higher order logic, one can state the existence of a bijection; (2) one can use multisets, a collection of unordered lists of elements where multiple instances can occur; and (3) one can define a permutation as the smallest equivalence relation containing the transpositions (i.e., the exchanges of elements). In [9,11], it is shown that the third approach is the best solution for automated proofs of the permutation property, but it is still not easy to define it formally. The literature contains various examples of permutation proofs, following the third approach [3,10,11,22,24,25]. In these papers, a permutation is formally defined and some of its properties (e.g., transitivity) are proved using deductive program verifiers, such as KeY [1] or Why3 [12] or interactive theorem provers like Coq [7]. However, they are ad hoc and there is a new proof for each algorithm.

In this paper, we recognize that there is a *uniform pattern* and we exploit this to prove the permutation property of any sequential and parallel swap-based sorting algorithm. We do this using VerCors [4], which is a deductive verification tool for reasoning about the correctness of concurrent programs. There are several advantages of our approach w.r.t. the previous work. First, none of the existing papers verified the permutation property of embarrassingly parallel sorting algorithms (e.g., in GPGPU). We demonstrate that our uniform approach also works for such algorithms by proving the permutation property of the parallel odd-even transposition sort algorithm. Second, the technique works for all languages supported in the tool such as C, Java and OpenCL, which means it is possible to prove the permutation property of sorting algorithms in a variety

of real-world languages. Third, in our permutation proof pattern, we use ghost variables[1] to keep track of value changes, which can be reused when establishing sortedness. Forth, we illustrate the generality of our approach by proving the permutation property of a vast collection of well-known sorting algorithms all together in one place.

Contributions. The main contributions of this paper are:

1. We outline a generic approach to verify the permutation property of any sequential and parallel swap-based sorting algorithm automatically.
2. We illustrate our technique by proving the permutation property of bubble sort, selection sort, insertion sort, parallel odd-even transposition sort, quick sort, two in-place merge sorts and TimSort, using the VerCors verifier.

Organization. Section 2 explains VerCors and its logic, by a verification example. Section 3 discusses the proposed generic approach to prove the permutation property and Sect. 4 applies it to an extensive collection of well-known sorting algorithms. Section 5 discusses related work and Sect. 6 concludes the paper.

2 VerCors

This section describes VerCors and the logic behind it along with a simple program verification example. VerCors[2] is a verifier to specify and verify (concurrent and parallel) programs written in a high-level language such as (subsets of) Java, C, OpenCL, OpenMP and PVL, where PVL is VerCors' internal language for prototyping new features. VerCors can be used to verify memory and thread safety and functional correctness of programs. The program logic behind VerCors is based on permission-based separation logic [2,5]. Programs are annotated with pre- and postconditions in permission-based separation logic. Permissions are used to capture which heap memory locations may be accessed by which threads, and verify memory and thread safety. Permissions are written as fractional values in the interval (0, 1] (cf. Boyland [6]): any fraction in the interval (0, 1) indicates a read permission, while 1 indicates a write permission.

Verification Example. Listing 1 shows a specification of a simple program that increments all the elements in an array by one. To specify permissions, we use predicates $Perm(L, \pi)$ where L is a heap location and π a fractional value in the interval $(0, 1]$[3]. Pre- and postconditions, (denoted by keywords `requires` and `ensures`, respectively in lines 2–4), should hold at the beginning and the end of the function, respectively. The keyword `context_everywhere` is used to specify a property that must hold throughout the function (line 1). As precondition, we have write permissions in all locations in the array (line 2). The postconditions

[1] A ghost variable is used for verification purposes and is not part of the program.
[2] See https://utwente.nl/vercors.
[3] The keywords `read` and `write` can also be used instead of fractions in VerCors.

List. 1. A simple sequential program

```
1  /*@ context_everywhere array != NULL && array.length == size;
2      requires (\forall* int k; k>=0 && k<size; Perm(array[k], write));
3      ensures (\forall* int k; k>=0 && k<size; Perm(array[k], write));
4      ensures (\forall int k; k>=0 && k<size; array[k] == \old(array[k])+1);
           @*/
5  void Inc(int[] array, int size) {
6
7    loop_invariant i>=0 && i<=size;
8    loop_invariant (\forall* int k; k>=0 && k<size; Perm(array[k], write));
9    loop_invariant (\forall int k; k>=0 && k<i; array[k] == \old(array[k])+1);
10   for(int i = 0; i < size; i++){
11       array[i] = array[i] + 1;
12 }
```

indicate first, we have write permissions in all locations in the array (line 3), and second, all values in the array are increased by one (line 4). Note that we use `\forall*` as universal separating conjunction over permission predicates and `\forall` as standard universal conjunction over logical predicates. Moreover, the keyword `\old` is used for an expression to refer to the value of that expression before entering a function (lines 4 and 9). The loop invariants specify that in each iteration we have write permissions to the array (line 8) and all values from index 0 up to index $i-1$ are increased by one (line 9). Then, the postcondition follows from these loop invariants.

3 Permutation Verification of Swap-Based Sorting

In this section, we describe our generic approach to verify the permutation property of sequential and parallel swap-based sorting algorithms.

3.1 Swap-Based Sorting Algorithms

An algorithm is a swap-based sorting algorithm if *by only swapping* the elements it satisfies the following:

- INPUT: An array *Input* of integers[4] of size N.
- OUTPUT: An array *Output* of integers of size N such that
  - sortedness: $\forall i.\ 0 \leq i < N-1$: $Output[i] \leq Output[i+1]$
  - permutation:
    $\forall i \in Input$: $occurrence(Input, i) == occurrence(Output, i)$.

where $occurrence(A, i)$ counts the number of occurrences of i in A.

[4] We specify the type of *Input* as integers, but it can be other types as well.

Algorithm 1. Sequential

```
1: invar: Output == inp_seq_cur
2: invar: Input == inp_seq_chain[0]
3: invar: properties to prove sortedness
4: loop(0 .. M)
       ...
5:     swap(Output, i, j)
6:     inp_seq_cur = swap-seq(inp_seq_cur,
7:                            i, j)
       ...
8:     inp_seq_chain = inp_seq_chain +
9:         seq<seq<int>> {inp_seq_cur}
10: end loop
```

Algorithm 2. Parallel

```
1: invar: Output == inp_seq_cur
2: invar: Input == inp_seq_chain[0]
3: invar: properties to prove sortedness
4: par(tid = 0.. K)
       ⋮
5:   swap(Output, f1(tid), f2(tid))
6:   atomic
7:     inp_seq_cur = swap-seq(
8:     inp_seq_cur, f1(tid), f2(tid))
9:   end atomic
       ⋮
10: end par
11: inp_seq_chain = inp_seq_chain +
12:     seq<seq<int>> {inp_seq_cur}
```

Fig. 1. Annotated pseudocode of sequential and parallel swap-based sorting algorithms.

3.2 Functional Correctness of Swap-Based Sorting Algorithms

To prove the correctness of swap-based sorting algorithms, we use ghost variables, in particular as sequences in VerCors. The most important advantage of using ghost variables is that it allows us to reason about *both* sortedness and permutation properties in the same specification. Moreover, establishing the proof based on ghost sequences helps us to also apply our technique on other data types rather than arrays such as linked lists. To demonstrate that, first we discuss which ghost variables we define and how they are beneficial in verifying sortedness. Then, we explain in detail how these ghost variables can be used to describe a generic verification pattern to verify the permutation property.

Figure 1 provides general sketches of the core (annotated) part of sequential and parallel swap-based sorting algorithms. Initially, we assume that *Input* and *Output* contain the same elements. The key operation in both sequential and parallel algorithms is the *swap* function where two elements are swapped. In the sequential algorithms, there is at most one swap at a time, but there might be multiple swaps in one iteration (e.g., sequential odd-even sort). In the parallel version there might be multiple simultaneous swaps, where f_1 and f_2 are two functions that assign a thread to two elements for swapping according to a thread id (i.e., *tid*). Note that in the parallel version, there can be a loop inside or outside the *par* block. By swapping the elements, a new rearrangement of the input array is generated. To keep track of these rearrangements of the elements, we define a ghost variable, *inp_seq_chain* (type sequence of sequences), as a chain of sequences that the first sequence in this chain is *Input*. We also define another ghost variable, *inp_seq_cur*, as the sequence that always stores the current rearrangement (i.e., last sequence in the chain).

List. 2. The *occurrence* function

```
1 /*@ ensures \result≥0 && \result≤|xs|;
2     ensures (\forall int i; 0≤i && i<|xs|; element != xs[i]) <==> \result
       == 0;
3     ensures (\forall int i; 0≤i && i<|xs|; element == xs[i]) <==>
4               \result == |xs|;
5     ensures element in xs <==> \result>0; @*/
6 static pure int occurrence(seq<int> xs, int element) = |xs| ≤ 0 ? 0 :
7     ( head(xs) == element ? (1+occurrence(tail(xs), element)) :
8       occurrence(tail(xs), element) );
```

Next, in the sequential version, we define a function, *swap-seq* and apply it to *inp_seq_cur* to update the current sequence exactly in the same way as the *swap* function does over *Output* (lines 6–7). Finally, we specify a loop invariant that shows that the array and the sequence are the same in each iteration (line 1). Moreover, in each iteration, after the swapping(s), we add the new current sequence to the chain of sequences (lines 8–9)[5]. This proposed pattern is also applicable for proving the correctness of recursive swap-based sorting algorithms (e.g., quick sort). In the parallel version, the principle is the same, but as there might be simultaneous swaps, we update the current sequence atomically (lines 6–9). Note that the exact location where we add the updated sequence to the chain depends on the algorithms and might be different from the sketches. For instance, if there is a loop inside the parallel block then we add the sequence to the chain at the end of the loop inside the parallel block. Notice that in the parallel algorithm (Fig. 1) the *swap* function over *Output* (line 5) is outside the atomic block. This matches for instance the parallel odd-even transposition sort. However, in other parallel sorting algorithms, we might need to include the *swap* function inside the atomic block to avoid data races[6]. Note that in the tool we use permission-based separation logic to prove data race-freedom of the algorithms.

These constructs allow us to prove the sortedness and permutation properties of any sequential and parallel swap-based sorting algorithm. By defining a chain of sequences, a user can provide key properties as invariants to reason about how the values change in the chain from *Input* to the last sequence (which is *Output*) to prove sortedness (line 3). To prove the permutation property, first we define a function *occurrence* (as shown in Listing 2) that counts the number of occurrences of an element in a sequence[7]. The postcondition of the function specifies (the boundary of) the result of the function in general (line 1) and in

[5] Note that depending on the algorithm, the new arrangement might be added to the chain after one swap *or* multiple swaps.

[6] A data race is a situation when two or more threads may access the same memory location simultaneously where at least one of them is a write.

[7] The `head` operation returns the first element of a sequence and `tail` returns a new sequence by eliminating the first element.

List. 3. The *permutation* function

```
1 static pure boolean permutation(seq<int> xs, seq<int> ys) = (|xs| == |ys|)
      &&
2   (\forall int i; 0≤i && i<|xs|; occurrence(xs, xs[i]) == occurrence(ys, xs[i
      ]));
```

three different conditions where the element exists in the sequence (line 5) or does not exist (lines 2) or the sequence only contains that element (lines 3–4).

Next, we define a predicate that states that a sequence is a permutation of another sequence if and only if the size of both are the same and the number of occurrences of each element in both are the same (Listing 3).

Next, we use VerCors to prove a property that for any sequence if we swap two arbitrary elements, the result is a permutation of the original sequence:

Property 3.1 For any sequence xs:
$(\forall i, j.0 \leq i \leq j < |xs| :$
$(\forall l.0 \leq l < |xs| : occurrence(xs, xs[l]) = occurrence(swap\text{-}seq(xs, i, j), xs[l])))$.

Proof. We define a lemma in VerCors to prove the property. We explain the steps that we have in the lemma (in VerCors) exactly as implemented[8].

If i equals to j both xs and $swap\text{-}seq(xs, i, j)$ are the same. Thus, the property holds and VerCors can infer it. If i is less than j, we split xs and $swap\text{-}seq(xs, i, j)$ into disjoint sequences in VerCors according to i and j as follows:

$$xs = xs[0..i-1] + xs[i] + xs[i+1..j-1] + xs[j] + xs[j+1..|xs|-1] \tag{1}$$

$$swap\text{-}seq(xs, i, j) = xs[0..i-1] + xs[j] + xs[i+1..j-1] + xs[i] + xs[j+1..|xs|-1] \tag{2}$$

We rewrite (1) and (2) in terms of the *occurrence* function and prove (by another lemma in VerCors) that this function distributes over concatenation of sequences as follows:

$$occurrence(xs + ys + ... + ts, element) = occurrence(xs, element) + occurrence(ys, element) + ... + occurrence(ts, element) \tag{3}$$

Then we note that we have the following equalities:

$$occurrence(xs, element) = occurrence(xs[0..i-1] + xs[i] + xs[i+1..j-1] + xs[j] + xs[j+1..|xs|-1], element) \tag{4}$$

[8] The full proof of all properties in VerCors is available at https://github.com/Safari1991/Permutation.

$$\begin{aligned} &\mathit{occurrence}(\mathit{swap\text{-}seq}(xs, i, j), \mathit{element}) = \mathit{occurrence}(xs[0..i-1] \\ &\quad + xs[j] + xs[i+1..j-1] + xs[i] + xs[j+1..|xs|-1], \mathit{element}) \end{aligned} \tag{5}$$

By applying property (3) to Eqs. (4) and (5) and using commutativity of "+" on integers, VerCors can conclude that both right-hand sides of (4) and (5) are equal, hence also their left-hand sides are equal. □

This allows us to specify that after each swap, the new sequence is a permutation of the previous one. As a corollary we prove that the *occurrence* function is symmetric:

Corollary 3.1. For any sequence xs:
$(\forall i, j.0 \leq i \leq j < |\mathit{swap\text{-}seq}(xs, i, j)| : (\forall l.0 \leq l < |\mathit{swap\text{-}seq}(xs, i, j)| :$
$\mathit{occurrence}(\mathit{swap\text{-}seq}(xs, i, j), \mathit{swap\text{-}seq}(xs, i, j)[l])$
$= \mathit{occurrence}(xs, \mathit{swap\text{-}seq}(xs, i, j)[l])))$.

Property 3.1 does not specify that the current sequence is a permutation of the input array (i.e., the first sequence in the chain). To establish that, we use VerCors to also prove that the *occurrence* function is transitive:

Property 3.2. For any equal-sized sequences xs, ys and ts:
$(\ (\forall l.0 \leq l < |xs| \rightarrow \mathit{occurrence}(xs, xs[l]) = \mathit{occurrence}(ys, xs[l])) \wedge$
$(\forall l.0 \leq l < |ys| \rightarrow \mathit{occurrence}(ys, ys[l]) = \mathit{occurrence}(ts, ys[l]))\ ) \Rightarrow$
$(\forall l.0 \leq l < |xs| \rightarrow \mathit{occurrence}(xs, xs[l]) = \mathit{occurrence}(ts, xs[l]))$

Proof. The proof is trivial using Corollary 3.1 and VerCors can infer this without intermediate proof steps. □

Using Properties 3.1 and 3.2 we can show that the permutation property is preserved for each new rearrangement of the input array during the algorithms:

Permutation Invariant. After each swap in the sequential and parallel swap-based sorting algorithms $\mathit{permutation}(\mathit{inp\_seq\_chain}[0], \mathit{inp\_seq\_cur})$ holds.

To understand why this is an invariant: (1) At the beginning, $\mathit{inp\_seq\_chain}[0]$ and $\mathit{inp\_seq\_cur}$ are equal to Input, hence the invariant holds. (2) assume that the invariant holds between sequences $\mathit{inp\_seq\_chain}[0]$ (which is Input) and $\mathit{inp\_seq\_chain}[M-1]$ (which is $\mathit{inp\_seq\_cur}$). Then, after each swap, we can apply Properties 3.1 and 3.2 to show that the invariant is preserved. Therefore, after the last swap when we add the updated $\mathit{inp\_seq\_cur}$ ($\mathit{inp\_seq\_chain}[M]$) to the chain, the invariant still holds. In the pseudocode of sequential and parallel swap-based sorting algorithms, we only need to apply the two properties before a swap. This is sufficient to prove the permutation property of the algorithms. Figure 2 illustrates this and completes the generic pattern. As we can see, the pattern can be generated automatically and the only parts that a user needs to fill out are the arguments i and j which are specific to the algorithms.

4 Case Studies: Proving Permutation of Swap-Based Sorting Algorithms

In this section, we show how we apply the technique described above, to verify multiple parallel and sequential swap-based sorting algorithms. In particular, we discuss how we prove the permutation property of parallel odd even transposition sort as well as sequential bubble sort, selection sort and insertion sort. Moreover, we illustrate how we use our approach to prove the permutation property of recursive in-place sorting algorithms quick sort and merge sort. In addition, we benefit from the verification of insertion sort and merge sort to verify the permutation property of TimSort[9].

Algorithm 1. Sequential

```
1:  invar: Output == inp_seq_cur
2:  invar: Input == inp_seq_chain[0]
3:  invar: properties to prove sortedness
4:  invar: permutation(inp_seq_chain[0],
5:                     inp_seq_cur)
6:  loop(0 .. M)
      ...
7:    swap(Output, i, j)
8:    applying Prop. 3.1 to inp_seq_cur
9:    applying Prop. 3.2 to
10:   inp_seq_chain[0], inp_seq_cur and
11:   swap-seq(inp_seq_cur, i, j)
12:   inp_seq_cur = swap-seq(inp_seq_cur,
13:                          i, j)
      ...
14:   inp_seq_chain = inp_seq_chain +
15:       seq<seq<int>> {inp_seq_cur}
16: end loop
```

Algorithm 2. Parallel

```
1:  invar: Output == inp_seq_cur
2:  invar: Input == inp_seq_chain[0]
3:  invar: properties to prove sortedness
4:  invar: permutation(inp_seq_chain[0],
5:                     inp_seq_cur)
6:  par(tid = 0.. K)
      ...
7:   swap(Output, f1(tid), f2(tid))
8:   atomic
9:     applying Prop. 3.1 to inp_seq_cur
10:    applying Prop. 3.2 to
11:    inp_seq_chain[0], inp_seq_cur and
12:    swap-seq(inp_seq_cur, f1(tid),
13:    f2(tid))
14:    inp_seq_cur = swap-seq(
15:    inp_seq_cur, f1(tid), f2(tid))
16:  end atomic
      ...
17:  end par
18: inp_seq_chain = inp_seq_chain +
19:     seq<seq<int>> {inp_seq_cur}
```

Fig. 2. Annotated pseudocode of sequential and parallel swap-based sorting algorithms.

4.1 Permutation Verification of Bubble, Selection and Insertion Sort

In this section, we use our generic pattern to verify the permutation property of bubble sort, selection sort, and insertion sort, as illustrated in Fig. 3. In bubble

[9] The full specifications of all case studies are available at https://github.com/Safari1991/Permutation.

Algorithm 3. Bubble sort

```
1: invar: Output == inp_seq_cur
2: invar: Input == inp_seq_chain[0]
3: invar: permutation(inp_seq_chain[0],
4:                    inp_seq_cur)
5: loop(k: 0 .. N-2)
6:   invar: Output == inp_seq_cur
7:   invar: Input == inp_seq_chain[0]
8:   invar: permutation(inp_seq_chain[0],
9:                      inp_seq_cur)
10:  loop(t: 0 .. N-k-2)
11:    if Output[t] > Output[t+1]
12:      swap(Output, t, t+1)
13:      applying Prop. 3.1 to inp_seq_cur
14:      applying Prop. 3.2 to
15:      inp_seq_chain[0], inp_seq_cur and
16:      swap-seq(inp_seq_cur, t, t+1)
17:      inp_seq_cur = swap-seq(
18:                    inp_seq_cur, t, t+1)
19:    end if
20:  end loop
21:  inp_seq_chain = inp_seq_chain +
22:    seq<seq<int>> {inp_seq_cur}
23: end loop
```

Algorithm 4. Selection sort

```
1: invar: Output == inp_seq_cur
2: invar: Input == inp_seq_chain[0]
3: invar: permutation(inp_seq_chain[0],
4:                    inp_seq_cur)
5: loop(k: 0 .. N-2)
6:   minIdx = k;
7:   invar: Output == inp_seq_cur
8:   invar: Input == inp_seq_chain[0]
9:   invar: properties to prove sortedness
10:  invar: permutation(inp_seq_chain[0],
11:                     inp_seq_cur)
12:  loop(t: k+1 .. N-1)
13:    if Output[t] < Output[minIdx]
14:      minIdx = t
15:    end if
16:  end loop
17:  swap(Output, k, minIdx)
18:  applying Prop. 3.1 to inp_seq_cur
19:  applying Prop. 3.2 to
20:  inp_seq_chain[0], inp_seq_cur and
21:  swap-seq(inp_seq_cur, k, minIdx)
22:  inp_seq_cur = swap-seq(
23:                inp_seq_cur, k, minIdx)
24:  inp_seq_chain = inp_seq_chain +
25:    seq<seq<int>> {inp_seq_cur}
26: end loop
```

Algorithm 5. Insertion sort

```
1: invar: Output == inp_seq_cur
2: invar: Input == inp_seq_chain[0]
3: invar: properties to prove sortedness
4: invar: permutation(inp_seq_chain[0], inp_seq_cur)
5: loop(k: 1 .. N-1)
6:   invar: Output == inp_seq_cur
7:   invar: Input == inp_seq_chain[0]
8:   invar: properties to prove sortedness
9:   invar: permutation(inp_seq_chain[0], inp_seq_cur)
10:  loop(t: k ..1)
11:    if Output[t-1] > Output[t]
12:      swap(Output, t-1, t)
13:      applying Prop. 3.1 to inp_seq_cur
14:      applying Prop. 3.2 to inp_seq_chain[0], inp_seq_cur and
15:      swap-seq(inp_seq_cur, t-1, t)
16:      inp_seq_cur = swap-seq(inp_seq_cur, t-1, t)
17:    end if
18:  end loop
19:  inp_seq_chain = inp_seq_chain + seq<seq<int>> {inp_seq_cur}
20: end loop
```

Fig. 3. Proving permutation property of bubble sort, selection sort and insertion sort using the proposed pattern.

Algorithm 6. Quick sort

```
1: if low < high
2:   pivot = Output[high], idx = low-1
3:   invar: Output == inp_seq_cur
4:   invar: Input == inp_seq_chain[0]
5:   invar: permutation(inp_seq_chain[0], inp_seq_cur)
6:   loop(k: low .. high-1)
7:     if Output[k] ≤ pivot
8:       idx++
9:       swap(Output, idx, k)
10:      applying Prop. 3.1 to inp_seq_cur and applying Prop. 3.2 to
11:      inp_seq_chain[0], inp_seq_cur and swap-seq(inp_seq_cur, idx, k)
12:      inp_seq_cur = swap-seq(inp_seq_cur, idx, k)
13:    end if
14:  end loop
15:  swap(Output, idx+1, high)
16:  applying Prop. 3.1 to inp_seq_cur and applying Prop. 3.2 to
17:  inp_seq_chain[0], inp_seq_cur and swap-seq(inp_seq_cur, idx+1, high)
18:  inp_seq_cur = swap-seq(inp_seq_cur, idx+1, high)
19:  inp_seq_chain = inp_seq_chain + seq<seq<int>> {inp_seq_cur}
20:  pivotIdx = idx+1
21:  recursive call for Output[low ... pivotIdx-1]
22:  recursive call for Output[pivotIdx+1 ... high]
23: end if
```

and insertion sort, there are two nested loops and a swap happens inside the inner loop. In selection sort, there are also two nested loops, but a swap happens in the outer loop. In all three algorithms, we follow exactly the approach as discussed in Sect. 3. We have the same invariants (for both loops) and we apply the same properties before a swap. The only differences are the two locations of elements to be swapped, which we set according to the algorithms themselves.

4.2 Permutation Verification of Quick Sort

The proposed pattern can also be used for recursive in-place sorting algorithms. To show this, we use the pattern to verify the permutation property of the quick sort algorithm as illustrated in Algorithm 6. This recursive algorithm is initialized with $low = 0$ and $high = |Output| - 1$. Each recursive call puts the last element (the pivot), in the correct position in the sorted array in such a way that all smaller elements will be to the left of the pivot and all larger elements will be to the right of the pivot. The function recursively applies the same function to both subarrays to the left and right of the pivot (lines 23–24), resulting in a sorted array. As we can see there are two swaps, one inside the loop and the other one outside the loop (lines 9 and 16). Again, we apply the properties before each swap and we add the new sequence (i.e., *inp_seq_cur*) to the chain (i.e., *inp_seq_chain*) after the second swap (line 21). In this way, we prove permutation of the quick sort algorithm.

Algorithm 7. In-place merge1

```
1: start2 = mid+1
2: if Output[mid] ≤ Output[start2]
3:   return
4: end if
5: invar: Output == inp_seq_cur
6: invar: Input == inp_seq_chain[0]
7: invar: permutation(inp_seq_chain[0],
8:                    inp_seq_cur)
9: while(start ≤ mid && start2 ≤ right)
10:   if Output[start] ≤ Output[start2]
11:     start+=1
12:   else
13:     idx = start2
14:     while(idx != start)
15:       swap(Output, idx-1, idx)
16:       applying Prop. 3.1 to inp_seq_cur
17:       applying Prop. 3.2 to
18:       inp_seq_chain[0], inp_seq_cur and
19:       swap-seq(inp_seq_cur, idx-1, idx)
20:       inp_seq_cur = swap-seq(inp_seq_cur,
21:                              idx-1, idx)
22:       idx−=1
23:     end while
24:     start+=1, mid+=1, start2+=1
25:   end if else
26: end while
27: inp_seq_chain = inp_seq_chain +
28:     seq<seq<int>> {inp_seq_cur}
```

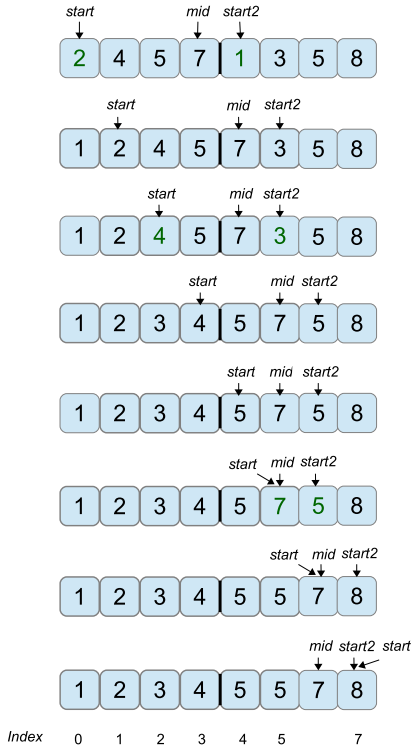

Fig. 4. Annotated pseudocode of an in-place merging (left) and an example (right). Green values in the example indicates that the comparison implies some swaps. (Color figure online)

4.3 Permutation Verification of Merge Sort

Merge sort is another example of a recursive sort that splits the elements into smaller parts (recursively) and merges them into a sorted array. Thus, the main part of the algorithm is merging two sorted subarrays. Figure 4 presents the (annotated) pseudocode and an example of an in-place merging. The example shows how the merge operates on two sorted subarrays (from indices 0–3 and 4–7). Initially, *start* points to the first element in the array (i.e., index 0), and *right* indicates the last element (i.e., index $N-1$). Moreover, the variable *start*2 points to the first location in the right subarray (i.e., index $mid+1$) where *mid* equals $(start+right)/2$. Then, the elements in these two locations are compared and if $Output[start] > Output[start2]$, we should insert the element in location *start*2 into location *start* by shifting all elements in between by one location

to the right (lines 12–25 of the pseudocode). Otherwise, we increase *start* by 1 (line 11). This process repeats until the array is sorted. As we can see in the pseudocode, the shifting is implemented as swapping two adjacent elements from location *start*2 to *start* consecutively. In this way, we can reuse the proposed pattern again to verify the permutation property of this algorithm.

Algorithm 8. In-place merge2

```
 1: if left==mid || mid==right
 2:   return
 3: end if
 4: start = mid-1, end = mid
 5: while(left≤start && end<right &&
 6:         Output[start]>Output[end])
 7:   start−=1, end+=1
 8: end while
 9: invar: Output == inp_seq_cur
10: invar: Input == inp_seq_chain[0]
11: invar: permutation(inp_seq_chain[0],
12:                    inp_seq_cur)
13: loop(k: 0 .. end-mid-1)
14:   swap(Output, 2×mid-end+k, mid+k)
15:   applying Prop. 3.1 to inp_seq_cur
16:   applying Prop. 3.2 to inp_seq_chain[0],
17:   inp_seq_cur and swap-seq(inp_seq_cur,
18:                  2×mid-end+k, mid+k)
19:   inp_seq_cur = swap-seq(inp_seq_cur,
20:                  2×mid-end+k, mid+k)
21: end loop
22: recursive call for
23: Output[left ... start+1 ... mid-1]
24: recursive call for
25: Output[mid ... end ... right-1]
```

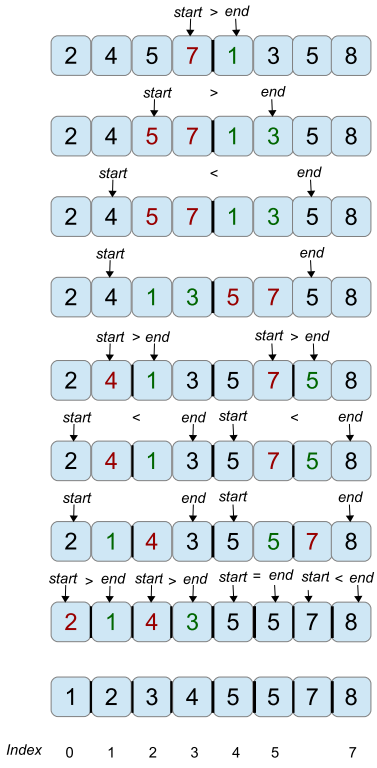

Fig. 5. Another in-place merging: annotated pseudocode (left) and an example (right). Red values should be swapped with green values in each recursion. (Color figure online)

Figure 5 presents another (annotated) pseudocode and an example of an in-place merge [8,17] which is more efficient than the previous one in complexity. In this algorithm, initially *left* points to index zero, *right* equals N and the two variables, *start* and *end* point to the last and first elements of the subarrays, respectively[10]. Variable *start* decreases and *end* increases by one until $Output[start] <= Output[end]$ (lines 5–8). When $Output[start] <= Output[end]$, we should swap the two subarrays in ranges (*start*, *mid*) (as red

[10] In the example, the middle element initially is in index 4, because *right* equals N.

Algorithm 9. In-place TimSort

```
1: RUN = 64, k=0
2: invar: Output == inp_seq_cur
3: invar: Input == inp_seq_chain[0]
4: invar: permutation(inp_seq_chain[0], inp_seq_cur)
5: while(k < N)
6:     end = min(k+RUN-1, N-1)
7:     insertion sort(Output[k ... end]
8:     k+=RUN
9: end while
10: chunk = RUN
11: invar: Output == inp_seq_cur
12: invar: Input == inp_seq_chain[0]
13: invar: permutation(inp_seq_chain[0], inp_seq_cur)
14: while(chunk < N)
15:     left = 0
16:     invar: Output == inp_seq_cur
17:     invar: Input == inp_seq_chain[0]
18:     invar: permutation(inp_seq_chain[0], inp_seq_cur)
19:     while(left < N)
20:         mid = left+chunk-1, right = min(left+2×chunk-1, N-1)
21:         if(mid < N-1)
22:             merge1(Output, left, mid, right)
23:             // or merge2(Output, left, mid+1, right+1)
24:         end if
25:         inp_seq_chain = inp_seq_chain + seq<seq<int>> {inp_seq_cur}
26:         left+=2×chunk
27:     end while
28:     chunk×=2
29: end while
```

values in the example on the right) and [*mid*, *end*) (as green values in the example), as in line 13–21 of the pseudocode. As we can see, we do swaps one by one between the first elements in the two ranges, then between the second elements, and so on. As a result, all elements in the left subarray become smaller than all the elements in the right subarray. This means that the subarrays are now independent and the same process can be applied for both of them. Therefore, we recursively call this process for the two subarrays to sort the full array (lines 22–25). Thus, the merge function is also recursive in addition to the main function of merge sort. Since there are swaps in this algorithm, we reuse the generic pattern and verify the permutation property of this algorithm as well. The only point is that, since the merging function is recursive, we add the new rearrangement of the elements (i.e., *inp_seq_cur*) to the chain (i.e., *inp_seq_chain*) in the main function of merge sort instead of in the merge function itself.

4.4 Permutation Verification of TimSort

Amongst the sorting algorithms, insertion sort performs better in practice when the number of elements are small (e.g., 64). Merge sort performs well when the size of two subarrays is power of 2. TimSort [20] benefits from this as a combination of insertion and merge sort. Algorithm 9 presents a simplified (annotated) version of this algorithm. It first sorts small groups of elements (e.g., 64) as runs (lines 5–9). Then, the algorithm repeatedly merges these equal-size sorted runs using the merging function (lines 14–29). That means, in the first iteration, the algorithm merges each two consecutive runs into a larger size of runs, and it repeats merging for each two consecutive (larger) runs until the full array is sorted. Note that we can use both merge functions discussed above for TimSort.

Since we already proved permutation of insertion and merge sort, we can easily prove the permutation property of TimSort. In fact, we prove permutation of two (in-place) TimSort algorithms using the two verified merge functions.

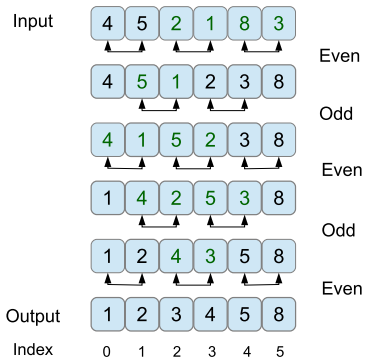

Fig. 6. An example of odd-even transposition sort. Values in green should be swapped. (Color figure online)

4.5 Permutation Verification of Parallel Odd-Even Transposition Sort

Odd-even transposition sort is a parallel version of bubble sort. It consists of two phases: odd and even. Algorithm 10 presents the annotated pseudocode while Fig. 6 shows an example of the execution of the algorithm. In the algorithm, *Output* is initialized to *Input*. In the even phase (lines 7–18), even locations ($2 \times tid$) are compared to their right neighbor ($2 \times tid + 1$) in line 8 and swapped if they are greater (line 9). In the odd phase (lines 20–31), odd locations ($2 \times tid + 1$) are compared and swapped in the same way with their right neighbor ($2 \times tid + 2$) in lines 21–22. This process repeats inside a loop (line 5) until all elements are sorted, i.e., there is no swap (indicated by a boolean, *isSorted*).

Algorithm 10. Parallel odd-even transposition sort

```
1: boolean isSorted = false;
2: invar: Output == inp_seq_cur
3: invar: Input == inp_seq_chain[0]
4: invar: permutation(inp_seq_chain[0], inp_seq_cur)
5: while !isSorted do
6:   isSorted = true;
7:   Par(tid = 0.. N/2) // thread id from 0 to N/2-1
8:     if 2 × tid + 1 < N && Output[2 × tid] > Output[2 × tid + 1] then
9:       Swap(Output, 2 × tid, 2 × tid + 1);
10:      atomic
11:        applying Property 3.1 to inp_seq_cur and applying Property 3.2 to
12:        inp_seq_chain[0], inp_seq_cur and swap-seq(inp_seq_cur, 2 × tid, 2 × tid + 1)
13:        inp_seq_cur = swap-seq(inp_seq_cur, 2 × tid, 2 × tid + 1)
14:      end atomic
15:      isSorted = false
16:    end if
17:  end par
18:  inp_seq_chain = inp_seq_chain + seq<seq<int>> {inp_seq_cur}
19:  Par(tid = 0.. N/2) // thread id from 0 to N/2-1
20:    if 2 × tid + 2 < N && Output[2 × tid + 1] > Output[2 × tid + 2] then
21:      Swap(Output, 2 × tid + 1, 2 × tid + 2);
22:      atomic
23:        applying Property 3.1 to inp_seq_cur and applying Property 3.2 to
24:        inp_seq_chain[0], inp_seq_cur and swap-seq(inp_seq_cur, 2 × tid + 1, 2 × tid + 2)
25:        inp_seq_cur = swap-seq(inp_seq_cur, 2 × tid + 1, 2 × tid + 2)
26:      end atomic
27:      isSorted = false
28:    end if
29:  end par
30:  inp_seq_chain = inp_seq_chain + seq<seq<int>> {inp_seq_cur}
31: end while
```

To prove permutation, we use the pattern proposed in Algorithm 2 (Fig. 2) for both phases. We only fill out the two locations that need to be swapped (i.e., $2 \times tid$ and $2 \times tid + 1$ in line 14, $2 \times tid + 1$ and $2 \times tid + 2$ in line 27). By applying the properties before a swap (lines 12–14 and 25–27), we establish the permutation property indicated as an invariant in line 4.

5 Related Work

There are several papers on proving sortedness and permutation properties of concrete sorting algorithms. In [10,22,24], the authors prove correctness of various sorting algorithms using the Why3 [12] platform[11]. Why3 is a program verifier which has its own language for programming and specification (i.e., WhyML)

[11] The verified sorting algorithms using Why3 are available at http://pauillac.inria.fr/~levy/why3/sorting.

based on first-order logic. It is mainly used as backend for other verifiers. To prove sortedness and permutation properties, suitable lemmas and invariants are defined and used in the extensive Why3 library[12]. However, they do not propose a generic approach to verify sortedness and permutation properties. Moreover, they do not verify any parallel sorting algorithms and they only prove the correctness of several sequential sorting algorithms.

Beckert et al. [3] prove JDK's dual pivot quick sort algorithm using KeY [1]. KeY is a program verifier for Java programs. The annotated Java programs are transformed into the internal dynamic logic representation of KeY, and then proof obligations are discharged to its first-order theorem prover which is based on sequent calculus. They benefit from sequences to prove sortedness and permutation properties of the algorithm. To prove the permutation property, they provide suitable invariants and prove some lemmas in the tool. They mention that proving the permutation property is by far the hardest part of their verification, which requires more interaction with the tool than the sortedness property. They neither outline a generic pattern nor verify any parallel sorting algorithms. In addition, they only verify quick sort in Java.

De Gouw et al. [13] found a bug in the TimSort implementation in one of OpenJDK's libraries while verifying the code using KeY. They show the effectiveness of (semi) automatic verification in finding bugs in a complex algorithm.

Filliâtre et al. [11] verify three sorting algorithms, insertion sort, quick sort and heap sort, in the Coq proof assistant. To prove the permutation property, they propose to express that the set of permutations is the smallest equivalence relation containing the transpositions (i.e., the exchanges of elements). We follow their approach to formally define permutation and prove its properties to prove any sequential and parallel swap-based sorting algorithms automatically.

Tushkanova et al. [25] discuss two specification languages, Java Modeling Language (JML) and Krakatoa Modeling Language (KML), to verify selection sort in Java automatically. To prove the permutation property, they use bags to show that the input and output array have the same content. Their approach is different from ours as we opt to not use bags to have a uniform pattern for verifying both sortedness and permutation.

6 Conclusion

Sorting algorithms are widely use in practice and their correctness is an important issue. To prove correctness of sorting algorithms, we should prove sortedness and permutation properties. Proving the permutation property is harder than sortedness, because it requires reasoning about existential quantifiers. In this paper, we propose a uniform approach to verify the permutation property of any sequential and parallel swap-based sorting algorithms. To demonstrate that our technique is generic, we prove the permutation property of bubble sort, selection sort, insertion sort, parallel odd-even transposition sort, in-place (recursive) quick sort, two merge sorts and TimSort using the VerCorse verifier.

[12] See http://why3.lri.fr/stdlib/array.html.

As future work, we plan to augment the proofs by providing the sortedness property for complex sorting algorithms such as odd-even transposition sort and TimSort using the proposed pattern.

References

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive Software Verification - The KeY Book, Lecture Notes in Computer Science, vol. 10001. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Amighi, A., Haack, C., Huisman, M., Hurlin, C.: Permission-based separation logic for multithreaded Java programs. LMCS **11**(1), 1–66 (2015)
3. Beckert, B., Schiffl, J., Schmitt, P.H., Ulbrich, M.: Proving JDK's dual pivot quicksort correct. In: Paskevich, A., Wies, T. (eds.) VSTTE 2017. LNCS, vol. 10712, pp. 35–48. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-72308-2_3
4. Blom, S., Darabi, S., Huisman, M., Oortwijn, W.: The VerCors tool set: verification of parallel and concurrent software. In: Polikarpova, N., Schneider, S. (eds.) IFM 2017. LNCS, vol. 10510, pp. 102–110. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66845-1_7
5. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. In: POPL, pp. 259–270 (2005)
6. Boyland, J.: Checking interference with fractional permissions. In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, pp. 55–72. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44898-5_4
7. The Coq proof assistant. https://coq.inria.fr/
8. Dvořák, S., Durian, B.: Merging by decomposition revisited. Comput. J. **31**(6), 553–556 (1988)
9. Filliâtre, J.C.: Deductive program verification. Ph.D. thesis, Université de Paris-Sud, vol. 11 (2011)
10. Filliâtre, J.C.: Deductive program verification with Why3 a tutorial (2013). https://www.lri.fr/~marche/DigiCosmeSchool/filliatre.html
11. Filliâtre, J.C., Magaud, N.: Certification of sorting algorithms in the Coq system. In: TPHOLs (1999). http://www-sop.inria.fr/croap/TPHOLs99/proceeding.html
12. Filliâtre, J.-C., Paskevich, A.: Why3—where programs meet provers. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8
13. 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: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 273–289. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_16
14. Govindaraju, N., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: high performance graphics co-processor sorting for large database management. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 325–336 (2006)
15. Greb, A., Zachmann, G.: GPU-ABiSort: optimal parallel sorting on stream architectures. In: PDP, pp. 10-pp. IEEE (2006)
16. Habermann, A.: Parallel Neighbor Sort. Computer Science Report. Carnegie-Mellon University, Pittsburgh (1972)

17. Kim, P.-S., Kutzner, A.: Stable minimum storage merging by symmetric comparisons. In: Albers, S., Radzik, T. (eds.) ESA 2004. LNCS, vol. 3221, pp. 714–723. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30140-0_63
18. Kipfer, P., Westermann, R.: Improved GPU sorting. GPU Gems **2**, 733–746 (2005)
19. Merrill, D.G., Grimshaw, A.S.: Revisiting sorting for GPGPU stream architectures. In: PACT, pp. 545–546 (2010)
20. Peters, T.: Timsort (2002). https://bugs.python.org/file4451/timsort.txt
21. Satish, N., Harris, M., Garland, M.: Designing efficient sorting algorithms for manycore GPUs. In: PDP, pp. 1–10. IEEE (2009)
22. Schoolderman, M.: Verification of Goroutines using Why3. Master's thesis, Institute for Computing and Information Sciences, RU Nijmegen (2016)
23. Sintorn, E., Assarsson, U.: Fast parallel GPU-sorting using a hybrid algorithm. J. Parallel Distrib. Comput. **68**(10), 1381–1388 (2008)
24. Tafat, A., Marché, C.: Binary heaps formally verified in Why3. Research Report RR-7780, INRIA (October 2011). https://hal.inria.fr/inria-00636083
25. Tushkanova, E., Giorgetti, A., Kouchnarenko, O.: Specifying and proving a sorting algorithm. Technical Report, University of Franche-Comte (October 2009). https://hal.archives-ouvertes.fr/hal-00429040