Software Verification 2014
Project Report

Roman Schmocker
09-911-215
December 5, 2014

1 Introduction

This report covers the implementation and verification of the two sort algorithms Bucketsort [4] and Quicksort [5]. The tools used to verify the code are AutoProof [1] and Boogie [6, 3].

The report is mostly structured based on my workflow during the project. The first section deals with the Boogie solution of the Quicksort and Bucketsort algorithm. The next section is about the AutoProof verification of both algorithms. The last section compares the two tools and draws conclusions.

2 Boogie

2.1 Quicksort

The Boogie solution for Quicksort contains the three procedures *`Swap`*, *`Split`* and *`QuickSort`*. All features manipulate a single global integer array where the sorting takes place.

The main pieces needed for verification are the postconditions of all three features, as well as the loop invariants in *`Split`*. The specifications can be roughly split into four different groups to verify that

- the result is sorted,
- the elements in the result array are the same as in the input,
- the procedure does not modify the global array outside the specified range,
- and the minimum and maximum values within a specified range stay the same.

The last point was necessary because, although Boogie could verify that all elements stay the same, it was not able to infer that the minimum and maximum values do not change as well. A workaround was therefore to keep track of these values explicitly by extending the interface of *`QuickSort`* to get the boundary values and add contracts which make sure the boundary is maintained.

Another challenge was to show that the elements of the final array are the same as in the input array. An extensive literature study [1] showed that the solution is to keep track of the permutation explicitly [2].

To do this I had to add two more arrays. Therefore the final solution ships with three global arrays:

[1] Typing "boogie sort algorithm" in Google.

- The initial input array. This array is never modified.
- The sorted output array.
- A permutation array to map the sorted array to the input array.

Using this system I was able to set up a "global" invariant that the permutation is always a valid mapping. This invariant can be found in basically every precondition, postcondition and loop invariant, and it's the reason why the permutation proof became very simple in the end.

2.2 Bucketsort

The Bucketsort implementation has three important procedures: `BucketSort`, `ConcatenateArray` and `MagicSort`. The last one defines a sort algorithm that exists as a specification only and is used by `BucketSort` to sort its buckets.

Due to time constraints I didn't attempt to prove that the result of the `BucketSort` procedure is a permutation of the input. Instead I just focused on proving that it is sorted.

The core of the algorithm is the loop that distributes the elements among three different buckets. It uses the following formula to determine the bucket an element $elem$ has to be placed in:

```
bucket_index := (elem + 3*N) div (2*N);
```

The loop invariant then basically states that all elements in the bucket with index i are in the range:

$$2iN <= elem + 3N < (2(i+1)N)$$

With this loop invariant Boogie was able to prove that all values in the "left" bucket are smaller than all values on the "middle" and "right" buckets, which it then could use to verify that the final array is sorted after calling `MagicSort` on each bucket.

3 Eiffel

3.1 Implementation

The implementation of the features in `SV_LIST` was pretty straightforward. The problem however was, as I later realized, that the implementation wasn't always suited to verification. Therefore the relevant sort features like Quicksort and Bucketsort had to be adapted several times and the final implementation carries some overhead to keep track of state needed for verification.

3.2 Specification

Due to my experience with Eiffel programs, specifying pre- and postconditions was easy. The only challenge were the sorting algorithms, where the specification should not just include that the array is sorted, but also that the final array contains the same elements as the input array. Otherwise one could just return a zero-filled array as a valid result. I solved this the same way as in the Boogie solution, by specifying that the result is a permutation of the input array.

The verification phase however brought some unplanned changes again. I had to add some specifications, such as the number of elements staying the same during a sort, as AutoProof wasn't able to infer that from the permutation fact only.

3.3 Verification

3.3.1 General featuers

The class *SV_LIST* was shipped with many small features such as *extend* or *at*. These were very easy to prove correct, often not even requiring any further checks or contracts besides pre- and postconditions.

3.3.2 Quicksort

The Quicksort algorithm is fully verified in AutoProof. Besides proving that the output is sorted, I was also able to prove that the final result is a permutation of the input array.

The algorithm selects the first element in the input as a pivot and then distributes the remaining elements into two arrays (*left* and *right*). It then recursively calls *quick_sort* on both elements and merges the two results.

At the core of the verification for Quicksort are the loop invariants during the split phase. There are two groups of invariants. The first group is required to verify the *sorted* postcondition. It basically states that all elements smaller than the pivot element are in the *left* array, whereas the other ones are in the *right* array.

The second group of invariants keeps track of the *permutation* postcondition. To be able to verify permutation I had to introduce a new array *control*, where all elements from the input are just inserted one by one. The invariants then state that the concatenation of *left*, *right* and the pivot element is a permutation of the *control* array, and that the *control* array will eventually be equal to the input array.

The main problem that I had during development was that AutoProof had some peculiar ideas about what could be a permutation. At some point I was able to verify that the input array is *equal* to the *control* array, and the *control* array is a permutation of the sorted array. AutoProof however couldn'd draw the conclusion that therefore the sorted array is a permutation of the input array. Thanks to a tip from Julian Tschannen I could finally resolve the issue by using a slightly different check for equality between the input and the *control* array.

I also encountered the same problem in AutoProof which I already had in Boogie: Even though I could prove that two arrays contain the same elements (using permutations), the tool was not able to infer that the range of possible values stays the same. To solve this I implemented the workaround from Boogie in Eiffel as well. This means that the Quicksort implementation gets the boundary values as an argument and has to make sure that they're maintained throughout the call.

Overall, the verification of Quicksort in AutoProof was very tough. In fact during my first attempt I stopped with the AutoProof solution and started working on Boogie. After that I had a slightly better idea about what's going on in the background, and I was finally able to proof the algorithm correct.

3.3.3 Bucketsort

After a successful verification of Quicksort, implementing and verifying Bucketsort was very easy. The algorithm is very similar to Quicksort. It also consists of splitting up the array into disjoint parts and then calling a sort algorithm on them. In this case the function didn't even have to be recursive, which simplified things a little.

I encountered one problem when trying to verify that the array is still sorted after calling *`quick_sort`* on each of the buckets. The solution was to introduce a new **`check`** instruction which states that the ranges of values within each bucket are still disjoint. This apparently helped the Boogie backend to trigger the right axiom, such that it was able to verify the algorithm.

4 Conclusion

During this project I was able to verify the Quicksort and Bucketsort algorithms in both AutoProof and Boogie. In my opinion Boogie is a little bit simpler to use because the language is fully designed with verification in mind and it's easier to figure out what is going on during the verification process.

AutoProof on the other hand has a higher level of abstraction. However I think that exactly this abstraction makes it harder to use AutoProof, because when something goes wrong you often have no idea and no hint about what causes the problem. Another problem which mostly happened in AutoProof is that I often got a timeout error, which can be very frustrating.

An advantage of AutoProof on the other hand is that it nicely integrates into the Eiffel programming language. This allows to at least partially prove some classes and features in a convenient and mostly automatic way and use them in real software, instead of just verifying a sort algorithm for a student project.

In both tools there's very little help provided when debugging a failed assertion. Getting it to verify therefore often involves hours of trial-and-error programming. I think for a static verifier to succeed in practice it needs to at least provide a counterexample or say why it couldn't verify a statement, but this might be infeasible to do.

Overall I think that static verification of programs is not yet very usable in practice, because it can be very hard to prove even simple programs which programmers can easily reason about. There's also the problem that the specification itself may be wrong with respect to the requirements, and the verifier thus happily verifies a buggy program. Therefore I think verification can at most serve as a complement to traditional software quality assurance techniques.

References

[1] Autoproof official website. http://se.inf.ethz.ch/research/autoproof/.

[2] Boogie bubblesort example. http://rise4fun.com/Boogie/Bubble.

[3] Boogie web interface. http://rise4fun.com/Boogie.

[4] Bucketsort. http://en.wikipedia.org/wiki/bucketsort.

[5] Quicksort. http://en.wikipedia.org/wiki/quicksort.

[6] M. Leino K. Rustan. This is boogie 2, 2008. http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf.