Software Verification Project
- AS 2015 -

Roger Koradi, Samuel Ueltschi
ETH Zürich, Switzerland

November 26, 2015

Abstract

This paper documents and discusses the authors' solutions to a project accompanying the ETH's *Software Verification* course in autumn semester 2015.

1 Introduction

Producing provably correct programs is becoming more and more important as society is being automatised and software is written that has control over safety-critical components, e.g. a car's brakes.

Proving a program correct is a tedious task, which leads to both, a high demand for tools assisting such proofs and an increased interest in research driving the development of these tools. The ETH's master course in *Software Verification* is - at the time of writing - encouraging students taking the course to complete a couple tasks requiring them to experiment with two notable verification tools: Autoproof[1] for Eiffel and Boogie[2] for the verification language with the same name.

In a first part of the project, the students are asked, this year, to complete an Eiffel program such that Autoproof can verify it. The program can be found in the appendix (**??**). We discuss our solution in section 2 and provide full code for our solution in **??**.

A second part of the project consists of modelling a sorting algorithm that alternates between quick- and bucketsort, depending on the elements in the array passed to it, in boogie. The appendix holds a more detailed description of the algorithm in form of a boogie template (**??**). We discuss our approach in section 3, along with some issues and interesting behaviours we came across and provide full code of our solution in **??**.

2 Autoproof

We are given a class

```
class
        SV_AUTOPROOF

feature
        lst: SIMPLE_ARRAY [INTEGER]
```

And we will specify its features below in a way such that Autoproof can verify them.
We refer to Appendix(**??**) for the complete code that we modify.

2.1 wipe

Wipe(**??**) takes an array of integers and resets all its item to 0 We add a loop invariant

```
across 1 |..| (k-1) as i all
        x.sequence [i.item] = 0 end
```

This is sufficient, because the first postcondition,

```
x.count = old x.count
```

is already maintained by another invariant.

2.2 mod_three

Procedure *mod_three* (**??**) takes two integer arrays *a,b* of equal length, uses of *wipe* on both and returns *b* with its every third element set to one.
First, we maintain that the amount of integers in *b* does not change:

```
b.count = b.count.old_
```

This invariant is necessary. Without it, the assignment

```
b[k] := a[k] + 1
```

may be out of bounds from Autoproof's point of view, because we iterate over the length of *a*, which is specified to be constant and only initially equal to the length of *b*.

Then, we need to postulate that each iteration over the loop can by itself change *b*:

```
modify(b)
```

Omitting this invariant will lead to Autoproof's insisting that *b* has never changed and that any further invariant claiming otherwise couldn't possibly be maintained.

Having specified that, we can now add an invariant

```
across 1 |..| (k-1) as i all
        (i.item \\ 3 = 0) implies
                b.sequence [i.item] = 1 end
```

which will claim that every third item we already iterated over in a is one in b. The assignment may read

```
b [k] := a[k] + 1
```

but the postcondition from *wipe(a)* allows Autoproof to deduce the element's being set to one without any further specification of ours.

2.3 swapper

Swapper(**??**) relies on *swap*(**??**) to reverse *lst* (which is global).

The loop here goes

```
from
        x := 1
        y := lst.count
until
        y <= x
```

and after each iteration, x is incremented by one and y is decremented by one. This allows us to use *y ¿ 0* as a way of specifying an invariant that trivially holds before *y := lst.count* has been executed and specifies some useful property afterwards - in our case, we use it to specify that once initialised, both *x* and *y* are within the bounds of *lst*, and therefore satisfy the precondition of *swap*.

```
y > 0 implies (1 <= x and x <= lst.count and 1 <= y
    and y <= lst.count)
```

For Autoproof to be able to proof that the swapped list is a permutation of the original list, we need to specify that all items not swapped remained the same. *Swap* itself does provide such a postcondition, however, this is insufficient because the *old lst swap*'s postcondition is mentioning is in fact the *lst* at the moment *swapper* is calling *swap*, which changes with each iteration. We must link these two *"olds"* explicitly:

```
across x |..| y as i all lst.sequence[i.item] =
        lst.sequence.old_[i.item] end
```

Swapper's postcondition states

```
across 1 |..| lst.count as i all lst.sequence [i.item]
    = (old lst.sequence) [lst.count - i.item + 1] end
```

This directly motivates the addition of the following two loop invariants:

```
x > 1 implies across 1 |..| (x-1) as i all lst.
    sequence[i.item] = lst.sequence.old_[lst.count-i.
    item + 1] end
```

```
x > 1 implies across 1 |..| (x-1) as i all lst.
    sequence[lst.count-i.item+1] = lst.sequence.old_[i.
    item] end
```

Where we are again using *x ¿ 1* as a way of saying "x and y have both been initialised". We need to split the postcondition into two parts because the items in-between x and y have not been swapped yet.

However, we are using *lst.count-i.item+1* so that it matches the postcondition and Autoproof cannot proof that just yet because it's never actually using this expression in the loop. What we thus require is another invariant that specifies the index the loop is using to correlate to the arithmetic in the postcondition and the invariants:

```
y = lst.count - x + 1
```

2.4 search

Search (**??**) is traversing *lst* backwards and returns *True* iff it contains search key v. We need to specify both, postconditions and invariants, so we start with the postconditions to help us find the invariants we need.

First, we specify that *lst*, who is required to be wrapped, will remain so.

```
lst.is_wrapped
```

Since *search* does not contain a *modify*-clause for *lst*, we do not need to explicitly specify that it does not change *lst*.

We then specify the actual return value:

```
Result implies across 1 |..| lst.count as i some lst.
    sequence[i.item] = v end
(not Result) implies across 1 |..| lst.count as i all
    lst.sequence[i.item] /= v end
```

We could have replaced these two implications by an equality

```
Result = across 1 |..| lst.count as i some lst.
    sequence[i.item] = v end
```

but not doing so allowed us to check each direction individually, which helped in finding the required invariants.

The loop here goes

```
from
        k := lst.count
        Result := False
until
        Result or k < 1
```

We first specify k to remain within the allowed range of indices for *lst*, using *k ¿ 0* to ignore its value prior to entering the loop and after exiting the loop when *lst* does not contain v:

```
k > 0 implies (1 <= k and k <= lst.count)
```

Because *Result* is only ever set when the current iteration finds v and k is only decremented when we do not, specifying the case where we do find v becomes easy:

```
Result implies (lst.sequence[k] = v)
```

To specify the case where we do not find v, we take into account that k is being decremented, starting from the last valid index down to zero, which means at any time during the iteration, all elements with a valid index greater than k have been checked not to be v.

```
(not Result) implies (across (k+1) |..| lst.count as i
     all lst.sequence[i.item] /= v end)
```

2.5 prod_sum

Prod_sum (**??**) was a less complicated matter. In fact, copying the postcondition proved sufficient:

```
zz * y + xx = xx.old_
```

2.6 paly

Paly (**??**) takes an integer array and returns *True* iff its elements form a palindrome (i.e. represent an integer string whose every prefix is a reversed suffix). We need to specify both, postconditions and invariants, and we start again with the postconditions to help us find the invariants:

```
Result implies across 1 |..| a.count as i all a.
    sequence[i.item] = a.sequence[a.count-i.item + 1]
    end
(not Result) implies across 1 |..| a.count as i some a
    .sequence[i.item] /= a.sequence[a.count-i.item+1]
    end
```

As with *search*, we could have expressed these two implications as a single equality but choose not to for easier reasoning about it.

The loop here goes

```
from
        x := 1
        y := a.count
        Result := True
until
        x >= y or not Result
```

We first specify x and y to remain within the bounds of the array, using y ¿ *0* to avoid violation on entry.

```
y > 0 implies (1 <= x and x <= a.count and 1 <= y and
    y <= a.count)
```

We then define a different interpretation for y to allow its translation into the arithmetic expected by the postconditions.

```
y = a.count - x + 1
```

Result is initialised to *True* and set to false as soon as we encounter a prefix that is not a reversed suffix. We exploit the fact that x is incremented with every iteration to express that *Result* = *True* implies "after at least one iteration" that the prefix iterated over so far is a reversed suffix:

```
(x > 1 and Result) implies across 1 |..| (x-1) as i
    all a.sequence[i.item] = a.sequence[a.count-i.item
    + 1] end
```

Because *Result* is iterated to true, we need no such trick for the other direction.

```
(not Result) implies across 1 |..| a.count as i some a
    .sequence[i.item] /= a.sequence[a.count-i.item + 1]
     end
```

3 Boogie

In this part we discuss the second part of the project, implementing and verifying quicksort and bucketsort in Boogie. First we describe how we chose to specify our algorithms eghow we described the complete behaviour of our sorting algorithms. Then we quickly discuss how the two algorithms were implemented followed by an in-depth discussion about the verification task where we describe the varous problems we faced and how we overcame them. Finally we compare the task of verifying and specifying in Boogie in contrast to Autoproof.

3.1 Algorithm Specification

Both quicksort and bucketsort basically share the same specification aside from some minor differences due to implementation details. Therefore we just state the specification of a general sorting algorithm in Boogie which applies to both quicksort and bucketsort.

The signature of our sort procedure looks as follows:

```
procedure xSort(lo: int, hi: int) returns (perm: [int]int)
  modifies a;
 ...
```

The sort procedure sorts a global variable `a:[int]int` from indices `lo` to `hi`. This global variable is a map which models a one-dimensional array of integers. We decided from the beginning to sort an arbitrary sub-sequence of the array because it would be useful when implementing quicksort. The return value of `xSort` is map from `int` to `int` which is a permutation on the array indices that describes how the elements of the array were swapped by `xSort`.

`sort` has only one precondition, namely that the input range must not be negative:

```
requires lo <= hi;
```

To specify the entire behaviour of the sort procedure, we state the following postconditions.

- After the procedure call, the array must be sorted:

  ```
  ensures (forall k, l: int ::
    lo <= k && k <= l && l <= hi ==> a[k] <= a[l]
  );
  ```

- The return value must be a valid permutation of the array indices.

```
ensures (
  forall i: int ::
    lo <= i && i <= hi
    ==> lo <= perm[i] && perm[i] <= hi
);
ensures (
  forall k, l: int ::
    lo <= k && k < l && l <= hi
    ==> perm[k] != perm[l]
);
```

- The final array is a permutation of the input array. Exactly the permuatation that is returned.

```
ensures (
  forall i: int ::
    lo <= i && i <= hi
    ==> a_qs[i] == old(a_qs)[perm[i]]
);
```

This specification is influenced by the bubblesort example that is given on the Boogie webpage [1]. We decided to copy the approach of using a concrete permutation to state that the final array is a permutation of the input array, because we felt that it would be easier to actually construct such a permutation rather than having to prove just the existance.

3.2 Quicksort Implementation

We implemented the textbook definition of quicksort as described in "Introduction to Algorithms". Quicksort is implemented by the `qs` procedure. It first partitions the array into a left and right part such that the left part is smaller than or equal to the pivot and the right part is greater than the pivot. The pivot element is always the right most element of the array. This partition step is implemented in a separate procedure called `qsPartition`. After partitioning, `qs` is called recursively on the left and the right part of the array. In addition to the actual sorting, `qs` and `qsPartition` both construct a permutation that keeps track on how the array elements were swapped. A big part of the implementation consists of code that combines the permutations returned by `qsPartition` and the two recursive calls to `qs` to a single permutation capturing all swap opperations.

[1] `http://rise4fun.com/Boogie/Bubble`

3.3 Bucketsort Implementation

Bucketsort is implemented in the `bucketSort` procedure. The procedure divides the array into three buckets with elements in the range of $[-3 * N)$, $[-N, N)$ and $[N, 3 * N)$ respectively. This is possible because we know that the procedure will only be called with elements in that range. The three buckets are then all sorted individually with quicksort. After that, the now sorted buckets are then copied back into the original array.

Just as in quicksort, the bucketsort implementation also has to construct a permutation that reflects the different swap operations. This was more challenging than in quicksort, because the size of the three buckets is not known in advance.
Note that being able to call quicksort with three buckets was not a trivial task. The problems are discussed in the next section.

3.4 Implementing modular sorting algorithm

In Boogie, as opposed to Eiffel/Autoproof, it isn't possible for procedures to modify arguments. This is why our sorting algorithm implementations both sort a global variable. In the case of bucketsort however it's necessary to sort three different arrays with the same procedure eġq̇uicksort. To solve this issue, we first modified our quicksort implementation to work on a different global variable `a_qs` and then added the following procedure:

```
procedure quickSort(arr : [int]int, lo : int, hi : int)
  returns (arr_sorted : [int]int, perm: [int]int)
  ...
{
  a_qs := arr;
  call perm := qs(lo,hi);
  arr_sorted := a_qs;
}
```

The new `quickSort` has the same specification as `qs` but instead of sorting a global array, it takes an array as argument and returns a sorted copy. This is achieved by using `a_qs` as temporary variable to store the input array. Now with this construction it is possible to sort arbitrary arrays using our existing quicksort implementation.

3.5 Verification

We managed to verify all our procedures with Boogie. However this was not an easy task. While working with boogie we encounterd several difficulties which will be discussed in this section.

A general problem that we faced in various places was the following. Given that a property could be proven to hold over a section of an array. Then a second while loop that modifies a different section of the same array would erase the previous information about the first section. We solved this problem by always adding all properties that had been previously shown about an array into the loop invariant of each subsequent while loop that would modify said array again. Unfortunately this lead to rather verbose loop invariants which would be repeated in several loops.

A variant of this problem also occured while verifying quicksort. Because each recursive call to `qs` modifies the same global variable `aqs`, all previous information about `aqs` are lost after ther procedure call. This was a problem because we would loose the information that the left part of the array is sorted after calling quicksort on the right part. This problem was solved by adding the following postcondition to `qs` and subsequently also `qsPartition`.

```
ensures (forall i: int :: i < lo || i > hi ==> a_qs[i] == old(a_qs)[i]);
```

The postcondition above states that only the elements in the range given by $[lo, hi]$ are modified. Proving this propery was straightforward and it resolved the problem that was described above.

Another big problem was the fact that the Boogie verifyier would not always terminate after a sensible amount of time. Furthermore, having the verifyier terminate could be influenced by adding certain assertions to the code. In fact, our final verification of quicksort requires two assertions to be present for the verifyer to terminate. We highlighted these assertions in our source code using comments.

The biggest problem we faced was during the verification of bucketsort. There we had to show that if a sorted array (a bucket) is copied into another array (the original array to sort), then this part of the array is also sorted. Proving this boiled down to the following lemma:

```
procedure lemma_absurde(a : [int]int, b:[int]int, off : int, n : int)
    requires off >= 0;
    requires n >= 0;
    requires (
      forall k: int :: 0 <= k && k < n
        ==> a[k+off] == b[k]
    );
```

```
    requires (
      forall k, l: int :: 0 <= k && k <= l && l < n
        ==> b[k] <= b[l]
    );
    ensures (
      forall k, l: int :: 0 <= k && k <= l && l < n
        ==> a[k+off] <= a[l+off]
    );
{
    assume (
      forall k, l: int :: 0 <= k && k <= l && l < n
        ==> a[k+off] <= a[l+off]
      );
}
```

`a` is the original array and `b` is the bucket array. The lemma states that if the bucket `b` is sorted and if the array `a` is equal to `b` in the index range from `off` to `off+n` then `a` is also sorted in this range. However boogie is not able to prove this lemma not even by assuming the postcondition in the procedure body.

In the end we figured out that the problem came from the way the array `a` was indexed. The lemma above uses a constant offset `off` to address a certain part in `a`. By reformulationg this problem into a form that doesn't use this offset, Boogie is able to prove the lemma:

```
procedure lemma(a : [int]int, b:[int]int, lo : int, hi : int, n : int)
    requires hi-lo+1==n;
    requires n >= 0;
    requires (forall k: int :: lo <= k && k <= hi ==> a[k] == b[k-lo]);
    requires (forall k, l: int :: 0 <= k && k <= l && l < n ==> b[k] <= b[l]);
    ensures (forall k, l: int :: lo <= k && k <= l && l <= hi ==> a[k] <= a[l]);
{ }
```

Figuring out this trick took us a long. Another problem we faced is the fact that Boogie is not always correct. We were able to have Boogie prove a false statement. Boogie would verify the following procedure:

```
const N: int;
axiom 0 <= N;
procedure gaga()
  ensures (forall k, l: int :: 0 <= k && k <= l && l < N ==> false);
{
  var x:int;
```

```
  var i :int;
  x := -N;
  i := 0;
  while(i < N) {
    i := i + 1;
  }
}
```

This turned out to be a bug which has been fixed in the meantime[2].
The various problems described above made the verification task rather frustrating compared to the first part of the project. In the next section we'll contrast the differences between AutoProof and Boogie.

3.6 Boogie vs. Autoproof

The biggest difference between AutoProof and Boogie is the level of abstraction that can be used to express statements. AutoProof offers a rich set of theories to reason about arrays in the form of sets, bags or sequences. With Boogie on the other hand, one can only reason about very low level constructs like maps.

Another difference between Boogie and AutoProof is the fact that it's not possible to modify arguments to a procedure whereas AutoProof eġĖiffel allows this. As consequence we had to come up with our own scheme to be able to sort arbitrary arrays.

In conclusion, we AutoProof to be a lot more conventient to reason about programs. Specifying and verifying a program in Boogie leads to a lot of verbose verification code because only low level constructs can be used to express properties whereas AutoProof already provides a range on conventient higher level constructs. Furthermore Boogie behaves strangely in many situations which can lead to a lot of frustration if one lacks experience.

References

[1] URL: `http://se.inf.ethz.ch/research/autoproof/`.

[2] K. Rustan M. Leino. *This is Boogie 2.* 2008. URL: `https://github.com/boogie-org/boogie`.

[2] `https://github.com/boogie-org/boogie/issues/25`