Software Verification Project
- AS 2015 -

Roger Koradi, Samuel Ueltschi
ETH Zürich, Switzerland

November 27, 2015

Abstract

This paper documents and discusses our solutions to a project accompanying the ETH's *Software Verification* course in autumn semester 2015, consisting of verifying two programs - one written in Eiffel and verified with AutoProof, one written and verified in Boogie - as well as he problems encountered as during the project and how we resolved the issues. Comparing our experiences with the two tools, we also briefly discuss why we preferred AutoProof over Boogie and why we still think that AutoProof may not be the answer to all program verification needs.

1 Introduction

Producing provably correct programs is becoming more and more important as society is being automatized 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[3] 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 (A.1). We discuss our solution in section 2 and provide full code for our solution in section 5.

A second part of the project consists of modeling 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 (A.2). 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 section 6.

In section 4, we then briefly summarise and discuss our experiences and impressions with Autoref and Boogie.

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(A.1) for the complete code that we modify.

2.1 wipe

Wipe(A.1.1) 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* (A.1.2) takes two integer arrays *a,b* of equal length, calls *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(A.1.4) relies on *swap*(A.1.3) 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 initialized, 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 what 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 (A.1.5) 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 (A.1.6) was a less complicated matter. In fact, copying the post-condition proved sufficient:

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

2.6 paly

Paly (A.1.7) 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 initialized 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 specified our algorithms i.e. how we described the complete behaviour of the two sorting algorithms. We then quickly discuss how the two algorithms were implemented followed by a discussion about the verification task. There we describe the various problems we faced and how we overcame them.

3.1 Algorithm Specification

Both quicksort and bucketsort 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` (both, `lo` and `hi`, inclusive). This global variable is a map which models a one-dimensional array of integers. We decided to specify that the sorting algorithm works on an arbitrary sub-sequence of the array because it results in a more general algorithm and because it makes implementing quicksort's recursivitiy easier. The return value of `xSort` is a map from `int` to `int` which represents a permutation on the array indices that describes how the elements of the array were swapped by `xSort`.

`xSort` 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 (from `lo` to `hi`).

```
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. This permutation is given by the return value `perm`.

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

This specification is based on the bubblesort example that is given on the Boogie web page [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 the existence of one.

3.2 Quicksort Implementation

We implemented the textbook definition of quicksort as described in "Introduction to Algorithms"[2]. Quicksort is implemented in the `qs` procedure. The pivot element is set to be the rightmost element of the array.
First the algorithm 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.
This partition step is implemented in a separate procedure `qsPartition`.
After partitioning, `qs` is called recursively on the left and the right part of the array. Here the fact that we specified the sorting procedures to work on arbitrary parts of the array comes in handy.

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

In addition to the actual sorting, `qs` and `qsPartition` both construct a permutation that keeps track of how the array's elements are swapped. A big part of the implementation consists of code that combines the permutations returned by `qsPartition` and the two recursive calls to `qs` into a single permutation capturing all swap operations.

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 $[-3N, -N)$, $[-N, N)$ and $[N, 3N]$ respectively. We decided to use these ranges because we know that the procedure will only be called with elements from $[-3N, 3N]$ and given no further information about them, we can but optimistically assume they are uniformly distributed over that range.
These three buckets are then each sorted individually using quicksort.
The now sorted buckets are then copied back into the original array yielding a sorted version of the original array.

Just as in quicksort, the bucketsort implementation also has to construct a permutation that reflects the swap operations of the algorithm. 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 because of some limitations of the Boogie language. This is addressed in the next section (3.4).

3.4 Implementing modular sorting algorithm

In Boogie, as opposed to Eiffel/AutoProof, it is not possible for procedures to modify input arguments. This is why our sorting algorithm is specified to work on a global variable. In the case of bucketsort however it is necessary to sort three different arrays with the same algorithm (quicksort). This however is not possible because the `qs` procedure can only sort one global variable (`a`). To solve this issue, we first modified our quicksort implementation to work on a different global variable `a_qs` and then created 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.

An alternative approach would have been to directly modify the `qs` implementation to take an array as argument and return a sorted copy of it. We decided against this solution because it would have required additional code that takes care of copying the values from input to output parameters. This would then add additional complexity to the verification task of the project. Our solution has the advantage that it allowed us to reuse our existing quicksort implementation with only minor modifications.

3.5 Verification

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

A general problem that we faced in various places was the following:
Assume that after a while loop, we have proven a property to hold for a section of an array. A second while loop that modifies a different section of the same array would then erase this 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 occurred while verifying quicksort.
Because each recursive call to `qs` modifies the same global variable `a_qs`, all previous information about `a_qs` are lost after the 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. We solved the issue 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 property was straightforward and it resolved the problem that was described above. Using this information, Boogie was able to infer that all previous properties about different parts of

the array still hold.

Another big problem was the fact that the Boogie verifier would not always terminate after a sensible amount of time. Furthermore, having the verifier 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 verifier 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 lemma1(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]
      );
}
```

Here `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 reformulating this problem into a form that doesn't use this offset, Boogie is able to prove the lemma:

```
procedure lemma2(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 time.

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 since been fixed[2].

The various problems described above made the verification task rather frustrating compared to the first part of the project. In section 4 we'll discuss the differences between working with AutoProof and Boogie.

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

4 Conclusion

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 and simple variables.

Another difference between Boogie and AutoProof is the fact that it is not possible to modify input arguments in a procedure, whereas AutoProof/Eiffel allows this. This makes writing reusable code much easier. To get the same effect in Boogie we had to come up with our own scheme to be able to sort arbitrary arrays.

In conclusion, we found AutoProof to be a lot more convenient to reason about programs with. 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 of convenient higher level constructs. Furthermore Boogie behaves strangely in many situations which can lead to a lot of frustration if one lacks experience with the tool.

Outside of our project, Boogie's demerit of being low-level can, however, be turned into the advantage of being very adaptive. If we wanted to write a verifier for a language different from Eiffel or Boogie, using Autoproof would force us to translate our program to Eiffel, first. Depending on the language, this may not even be possible because certain constructs are not expressible in Eiffel. Given that Autoproof itself is translating the Eiffel programs passed to it to Boogie[4], such a restriction would be a mere waste of expressiveness and we would be better advised to just directly translate to Boogie.

References

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

[2] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

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

[4] Julian Tschannen et al. *Verifying Eiffel Programs with Boogie*. Tech. rep. ETHZ.

5 Eiffel Solution Code

```
class
    SV_AUTOPROOF

feature
    lst: SIMPLE_ARRAY [INTEGER]
```

5.0.1 wipe

```
feature
wipe (x: SIMPLE_ARRAY [INTEGER])
note
        explicit: wrapping
require
        x /= Void
        modify (x)
local
        k: INTEGER
do
        from
                k := 1
        invariant
                x.is_wrapped
                x.count = x.count.old_
                across 1 |..| (k-1) as i all x.
                   sequence [i.item] = 0 end
        until
                k > x.count
        loop
                x [k] := 0
                k := k + 1
        end
ensure
        x.count = old x.count
        across 1 |..| x.count as i all x.sequence [i.
           item] = 0 end
end
```

5.1 mod_three

```
mod_three (a, b: SIMPLE_ARRAY [INTEGER])
note
        explicit: wrapping
require
        a /= Void
        b /= Void
        a /= b
        a.count = b.count
        a.count > 0
        modify (a, b)
local
        k: INTEGER
do
        wipe (a)
        wipe (b)
        from
                k := 1
        invariant
                a.is_wrapped and b.is_wrapped
                a.count = a.count.old_
                modify(b)
                b.count = b.count.old_
                across 1 |..| (k-1) as i all (i.item
                   \\ 3 = 0) implies b.sequence [i.
                   item] = 1 end
        until
                k > a.count
        loop
                if k \\ 3 = 0 then
                        b [k] := a[k] + 1
                end
                k := k + 1
        end
ensure
        across 1 |..| b.count as i all (i.item \\ 3 =
           0) implies b.sequence [i.item] = 1 end
end
```

5.2 swap

```
feature
swap (x, y: INTEGER)
note
        explicit: wrapping
require
        lst.is_wrapped
        1 <= x and x <= lst.count
        1 <= y and y <= lst.count
        modify (lst)
local
        z: INTEGER
do
        z := lst [x]
        lst [x] := lst [y]
        lst [y] := z
ensure
        lst.is_wrapped
        lst.count = old lst.count
        across 1 |..| lst.count as i all i.item /= x
           and i.item /= y implies lst.sequence [i.
           item] = (old lst.sequence) [i.item] end
        lst.sequence [x] = (old lst.sequence) [y]
        lst.sequence [y] = (old lst.sequence) [x]
end
```

5.3 swapper

```
swapper
note
        explicit: wrapping
require
        lst.is_wrapped
        lst /= Void
        modify (lst)
local
        x, y: INTEGER
do
        from
                x := 1
                y := lst.count
        invariant
                lst.is_wrapped
                lst.sequence.count = lst.sequence.old_
                    .count

                y > 0 implies (1 <= x and x <= lst.
                    count and 1 <= y and y <= lst.count
                    )
                across x |..| y as i all lst.sequence[
                    i.item] = lst.sequence.old_[i.item]
                     end
                        -- necessary because the "old"
                            are out of sync (swapper's
                            "old" is different from
                           swap's)

                y = lst.count - x + 1
                        --necessary for the following
                            two invariants to succeed
                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

        until
                y <= x
```

```
        loop
                swap (x, y)
                x := x + 1
                y := y - 1
        end
ensure
        lst.is_wrapped
        lst.sequence.count = (old lst.sequence).count
        across 1 |..| lst.count as i all lst.sequence
           [i.item] = (old lst.sequence) [lst.count -
           i.item + 1] end
end
```

5.4 search

```
feature
search (v: INTEGER): BOOLEAN
note
        status: impure
require
        lst.is_wrapped
        lst /= Void
local
        k: INTEGER
do
        from
                k := lst.count
                Result := False
        invariant
                k > 0 implies (1 <= k and k <= lst.
                   count)
                Result implies (lst.sequence[k] = v)
                (not Result) implies (across (k+1)
                   |..| lst.count as i all lst.
                   sequence[i.item] /= v end)
        until
                Result or k < 1
        loop
                if lst [k] = v then
                        Result := True
                else
                        k := k - 1
                end
        variant
                k - if Result then 1 else 0 end
        end
ensure
        lst.is_wrapped
        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
end
```

5.5 prod_sum

```
feature
xx, zz: INTEGER

prod_sum (y: INTEGER)
require
        xx >= 0
        zz >= 0
        y > 0
do
        from
                zz := 0
        invariant
                is_open
                inv
                zz * y + xx = xx.old_
        until
                xx < y
        loop
                zz := zz + 1
                xx := xx - y
        end
ensure
        zz * y + xx = old xx
end
```

5.6 paly

```
feature
paly (a: SIMPLE_ARRAY [INTEGER]): BOOLEAN
note
        explicit: wrapping
require
        a /= Void
local
        x, y: INTEGER
do
        from
                x := 1
                y := a.count
                Result := True
        invariant
                y > 0 implies (1 <= x and x <= a.count
                    and 1 <= y and y <= a.count)
                y = a.count - x + 1
                        --necessary for the following
                            invariant to succeed
                (x > 1 and Result) implies across 1
                   |..| (x-1) 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
        until
                x >= y or not Result
        loop
                if a [x] /= a [y] then
                        Result := False
                end
                x := x + 1
                y := y - 1
        variant
                y - x
        end
ensure
        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
end

end -- end of class
```

6 Boogie Solution Code

```
// Introduce a constant 'N' and postulate that it is
   non-negative
const N: int;
axiom 0 <= N;

// Declare a map from integers to integers.
// 'a' should be treated as an array of 'N' elements,
   indexed from 0 to N-1
var a: [int]int;
// internal variable that is used by quicksort
   implementation
var a_qs: [int]int;

// Returns true iff the elements of 'arr' are small (i
   .e. values in the range -3N to +3N)
function has_small_elements(arr: [int]int): bool
{
  (forall i: int :: (0 <= i && i < N) ==> (-3 * N <=
     arr[i] && arr[i] <= 3 * N))
}

// Sorts 'a' using bucket sort or quick sort, as
   determined by has_small_elements(a)
procedure sort() returns (perm : [int]int)
  modifies a, a_qs;
  // perm is a permutation
  ensures (forall i: int :: 0 <= i && i <= N-1 ==> 0
     <= perm[i] && perm[i] <= N-1);
  ensures (forall k, l: int :: 0 <= k && k < l && l <=
      N-1 ==> perm[k] != perm[l]);
  // the final array is that permutation of the input
     array
  ensures (forall i: int :: 0 <= i && i <= N-1 ==> a[i
     ] == old(a)[perm[i]]);
  // array is sorted
  ensures (forall k, l: int :: 0 <= k && k <= l && l
     <= N-1 ==> a[k] <= a[l]);
{

  if(N > 0) { // array not empty
    if (has_small_elements(a))
```

```
    {
        // sort 'a' using bucket sort
        call perm := bucketSort(0,N-1);
    } else {
        // sort 'a' using quick sort
        call a, perm := quickSort(a,0,N-1);
    }
  }

}

procedure qsPartition(lo : int, hi : int) returns (
   pivot_index: int, perm: [int]int)
  modifies a_qs;
  requires lo <= hi;
  ensures lo <= pivot_index && pivot_index <= hi;
  // array is correctly partitioned
  ensures (forall k: int :: lo <= k && k < pivot_index
      ==> a_qs[k] <= a_qs[pivot_index]);
  ensures (forall k: int :: pivot_index < k && k <= hi
      ==> a_qs[k] > a_qs[pivot_index]);
  // perm is a permutation
  ensures (forall i: int :: lo <= i && i <= hi ==> lo
     <= perm[i] && perm[i] <= hi);
  ensures (forall i, j: int :: lo <= i && i < j && j
     <= hi ==> perm[i] != perm[j]);
  // the final array is that permutation of the input
     array
  ensures (forall i: int :: lo <= i && i <= hi ==>
     a_qs[i] == old(a_qs)[perm[i]]);
  // only the indexes between lo and hi are modified,
     the rest of the array stays the same
  ensures (forall i: int :: i < lo || i > hi ==> a_qs[
     i] == old(a_qs)[i]);
{

  // local variables
  var i, j, pivot, tmp, n : int;

  // init permutation
  n := lo;
  while (n <= hi)
    invariant lo <= n && n <= hi+1;
    invariant (forall k: int :: lo <= k && k < n ==>
```

```
      perm[k] == k);
  invariant (forall k: int :: lo <= k && k < n ==>
      lo <= perm[k] && perm[k] <= hi);
  invariant (forall k, l: int :: lo <= k && k < l &&
      l < n ==> perm[k] != perm[l]);
{
  perm[n] := n;
  n := n + 1;
}

i := lo - 1;
j := lo;


pivot := a_qs[hi];
while(j < hi)
  invariant pivot == a_qs[hi];
  invariant lo - 1 <= i && i < j && j <= hi;

  // correct partition
  invariant (forall k: int :: lo <= k && k <= i ==>
      a_qs[k] <= pivot);
  invariant (forall k: int :: i < k && k < j ==>
      a_qs[k] > pivot);

  // correct permutation
  invariant (forall k: int :: lo <= k && k <= hi ==>
      lo <= perm[k] && perm[k] <= hi);
  invariant (forall k, l: int :: lo <= k && k < l &&
      l <= hi ==> perm[k] != perm[l]);

  // final array is permutation of original
  invariant (forall k: int :: lo <= k && k <= hi ==>
      a_qs[k] == old(a_qs)[perm[k]]);

  // rest of the array is untouched
  invariant (forall k: int :: k < lo || k > hi ==>
      a_qs[k] == old(a_qs)[k]);

{
  if(a_qs[j] <= pivot) {
    i := i + 1;
    // swap a_qs[i] with a_qs[j]
    tmp := a_qs[i]; a_qs[i] := a_qs[j]; a_qs[j] :=
```

```
            tmp;
        tmp := perm[i]; perm[i] := perm[j]; perm[j] :=
            tmp;
      }
      j := j + 1;
    }

    //swap a_qs[i+1] with a_qs[hi], the pivot element
    tmp := a_qs[i+1];  a_qs[i+1] := a_qs[hi];  a_qs[hi]
       := tmp;
    tmp := perm[i+1]; perm[i+1] := perm[hi]; perm[hi] :=
        tmp;

    pivot_index := i+1;

}

// public interface for quicksort that can be used to
    sort arbitrary arrays
procedure quickSort(arr : [int]int, lo : int, hi : int
    ) returns (arr_sorted : [int]int, perm: [int]int)
    modifies a_qs;
    requires lo <= hi;
    // perm is a permutation
    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 that permutation of the input
       array
    ensures (forall i: int :: lo <= i && i <= hi ==>
       arr_sorted[i] == arr[perm[i]]);
    // array is sorted
    ensures (forall k, l: int :: lo <= k && k <= l && l
       <= hi ==> arr_sorted[k] <= arr_sorted[l]);
{
    // write input array into a_qs
    a_qs := arr;
    // let quicksort implementation sort a_qs
    call perm := qs(lo,hi);
    // write a_qs into output argument
    arr_sorted := a_qs;
}
```

```
// sorts global array 'a_qs' in the index range [lo
    ,...,hi] (hi is inclusive)
procedure qs(lo : int, hi : int) returns (perm: [int]
    int)
  modifies a_qs;
  requires lo <= hi;
  // perm is a permutation
  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 that permutation of the input
     array
  ensures (forall i: int :: lo <= i && i <= hi ==>
     a_qs[i] == old(a_qs)[perm[i]]);
  // array is sorted
  ensures (forall k, l: int :: lo <= k && k <= l && l
     <= hi ==> a_qs[k] <= a_qs[l]);
  // rest of the array is untouched
  ensures (forall k: int :: k < lo || k > hi ==> a_qs[
     k] == old(a_qs)[k]);
{
   // local variables
  var n, pivot_index: int;
  var perm_comb, perm_left, perm_right, perm_res: [int
     ]int;

  if(lo < hi) {
    call pivot_index, perm := qsPartition(lo,hi);

    // we have a non empty left part
    if(lo < pivot_index) {
      call perm_left := qs(lo, pivot_index - 1);

      n := lo;
      while(n < pivot_index)
        invariant lo <= n && n <= pivot_index;
        invariant (forall i: int :: lo <= i && i < n
           ==> perm_comb[i] == perm_left[i]);
        invariant (forall i: int :: lo <= i && i < n
           ==> lo <= perm_comb[i] && perm_comb[i] <=
           hi);
        invariant (forall k, l: int :: lo <= k && k <
           l && l < n ==> perm_comb[k] != perm_comb[l
```

```
        ]);
    {
      perm_comb[n] := perm_left[n];
      n := n+1;
    }
  }

  // we have a non empty right part
  if(pivot_index + 1 <= hi) {
    call perm_right := qs(pivot_index + 1, hi);

    n := pivot_index + 1;
    while(n <= hi)
      // maintain previous information about
         perm_comb
        invariant (forall i: int :: lo <= i && i <
           pivot_index ==> perm_comb[i] == perm_left
           [i]);
        invariant (forall i: int :: lo <= i && i <
           pivot_index ==> lo <= perm_comb[i] &&
           perm_comb[i] <= hi);
        invariant (forall k, l: int :: lo <= k && k
           < l && l < pivot_index ==> perm_comb[k]
           != perm_comb[l]);
        invariant (forall i: int :: lo <= i && i <=
           pivot_index-1 ==> a_qs[i] == old(a_qs)[
           perm[perm_comb[i]]]);

        invariant pivot_index + 1 <= n && n <= hi+1;
        invariant (forall i: int :: pivot_index + 1
           <= i && i < n ==> perm_comb[i] ==
           perm_right[i]);
        invariant (forall i: int :: pivot_index + 1
           <= i && i < n ==> pivot_index + 1 <=
           perm_comb[i] && perm_comb[i] <= hi);
        invariant (forall k, l: int :: pivot_index +
            1 <= k && k < l && l < n ==> perm_comb[k
           ] != perm_comb[l]);
    {
      perm_comb[n] := perm_right[n];
      n := n+1;
    }
  }
```

```
    perm_comb[pivot_index] := pivot_index;
    // perm_comb is now a permutation

    // IMPORTANT: this assert is needed for boogie to
       terminate
    assert (forall k, l: int :: lo <= k && k <= l && l
        <= hi  ==> a_qs[k] <= a_qs[l]);

    // combine perm_comb with perm
    n := lo;
    while(n <= hi)
      invariant lo <= n && n <= hi+1;
      invariant (forall i: int :: lo <= i && i < n ==>
          perm_res[i] == perm[perm_comb[i]]);
      invariant (forall i: int :: lo <= i && i < n ==>
          lo <= perm_res[i] && perm_res[i] <= hi);
      invariant (forall k, l: int :: lo <= k && k < l
         && l < n ==> perm_res[k] != perm_res[l]);
      invariant (forall i: int :: lo <= i && i <= hi
         ==> a_qs[i] == old(a_qs)[perm[perm_comb[i]]])
          ;
    {
      perm_res[n] := perm[perm_comb[n]];
      n := n+1;
    }

    // write updated permutation back to perm
    perm := perm_res;

  } else {
    perm[lo] := lo;
  }

  // IMPORTANT: this assert is needed for boogie to
     terminate
  assert (forall i: int :: lo <= i && i <= hi ==> a_qs
     [i] == old(a_qs)[perm[i]]);
}


// sorts global array 'a' in the index range [lo,...,
   hi] (hi is inclusive)
procedure bucketSort(lo : int, hi : int) returns (perm
```

```
    : [int]int)
  modifies a,a_qs;
  // range must not be negative
  requires 0 <= lo && lo <= hi;
  // perm is a permutation
  ensures (forall k: int :: lo <= k && k <= hi ==> lo
     <= perm[k] && perm[k] <= hi);
  ensures (forall k, l: int :: lo <= k && k < l && l
     <= hi ==> perm[k] != perm[l]);
  // the final array is that permutation of the input
     array
  ensures (forall k: int :: lo <= k && k <= hi ==> a[k
     ] == old(a)[perm[k]]);
  // array is sorted
  ensures (forall k, l: int :: lo <= k && k <= l && l
     <= hi  ==> a[k] <= a[l]);
{

  //buckets' end indices
  var b0_i, off_b0:int;
  var b1_i, off_b1:int;
  var b2_i, off_b2:int;

  //buckets' upper bounds (exclusive)
  var bound_0:int;
  var bound_1:int;
  var bound_2:int;

  //bucket permutations
  var perm0, perm1, perm2 : [int]int;
  var permAtoB0 : [int]int;
  var permAtoB1 : [int]int;
  var permAtoB2 : [int]int;
  var permA : [int]int;

  var b0, b0_sorted, b1, b1_sorted, b2, b2_sorted: [
     int]int;

  //iterator variables
  var i :int;

  b0_i := 0;
  b1_i := 0;
  b2_i := 0;
```

```
bound_0 := -1*N;
bound_1 := N;
bound_2 := 3*N + 1;

i := lo;
while(i <= hi)
  invariant (i >= lo && i <= hi+1);
  // array 'a' is divided over b0,b1 and b2
  invariant i-lo == b0_i+b1_i+b2_i;
  //all buckets contain the correct kind of elements
     :
  invariant(forall e:int :: (e >= 0 && e < b0_i) ==>
      (b0[e] < -1*N));
  invariant(forall e:int :: (e >= 0 && e < b1_i) ==>
      (b1[e] >= -1*N && b1[e] < N));
  invariant(forall e:int :: (e >= 0 && e < b2_i) ==>
      (b2[e] >= N));

  //permAtoB holds the mapping of elements from A
     onto the buckets
  invariant(forall e:int :: (e >= 0 && e < b0_i) ==>
      (b0[e] == a[permAtoB0[e]]));
  invariant(forall e:int :: (e >= 0 && e < b1_i) ==>
      (b1[e] == a[permAtoB1[e]]));
  invariant(forall e:int :: (e >= 0 && e < b2_i) ==>
      (b2[e] == a[permAtoB2[e]]));

  invariant (forall k: int :: 0 <= k && k < b0_i ==>
      lo <= permAtoB0[k] && permAtoB0[k] <= hi);
  invariant (forall k: int :: 0 <= k && k < b1_i ==>
      lo <= permAtoB1[k] && permAtoB1[k] <= hi);
  invariant (forall k: int :: 0 <= k && k < b2_i ==>
      lo <= permAtoB2[k] && permAtoB2[k] <= hi);

  //every element is permuted
  invariant (forall k: int :: 0 <= k && k < b0_i ==>
      permAtoB0[k] < i);
  invariant (forall k: int :: 0 <= k && k < b1_i ==>
      permAtoB1[k] < i);
  invariant (forall k: int :: 0 <= k && k < b2_i ==>
      permAtoB2[k] < i);

  invariant(forall e,f : int :: ((e >= 0 && e < b0_i
      && e != f && f >= 0 && f < b0_i) ==> (
```

```
      permAtoB0[e] != permAtoB0[f])));
  invariant(forall e,f : int :: ((e >= 0 && e < b1_i
      && e != f && f >= 0 && f < b1_i) ==> (
      permAtoB1[e] != permAtoB1[f])));
  invariant(forall e,f : int :: ((e >= 0 && e < b2_i
      && e != f && f >= 0 && f < b2_i) ==> (
      permAtoB2[e] != permAtoB2[f])));

  invariant(forall e,f : int :: ((e >= 0 && e < b0_i
      && f >=0 && f < b1_i) ==> (permAtoB0[e] !=
      permAtoB1[f])));
  invariant(forall e,f : int :: ((e >= 0 && e < b0_i
      && f >=0 && f < b2_i) ==> (permAtoB0[e] !=
      permAtoB2[f])));
  invariant(forall e,f : int :: ((e >= 0 && e < b1_i
      && f >=0 && f < b2_i) ==> (permAtoB1[e] !=
      permAtoB2[f])));
{
  //add a[i] to correct bucket
  if(a[i] < bound_1){
    if(a[i] < bound_0){
      b0[b0_i] := a[i];
      permAtoB0[b0_i] := i;

      b0_i := b0_i + 1;
    }
    else{
      b1[b1_i] := a[i];
      permAtoB1[b1_i] := i;

      b1_i := b1_i + 1;
    }
  }
  else{
    b2[b2_i] := a[i];
    permAtoB2[b2_i] := i;

    b2_i := b2_i + 1;
  }
  i := i + 1;
}

i := 0;
while(i < b0_i)
```

```
  invariant (i >= 0 && i <= b0_i);
  invariant(forall e:int :: (e >= 0 && e < i) ==> (
     b0[e] == a[permA[e]]));
  invariant (forall k: int :: 0 <= k && k < i ==> lo
      <= permA[k] && permA[k] <= hi);

  invariant (forall e:int :: (e >=0 && e < i ==>
     permA[e] == permAtoB0[e]));
  invariant (forall e,f:int :: (e >= 0 && e < f && f
      < i ==> permA[e] != permA[f]));
{
  permA[i] := permAtoB0[i];
  i := i + 1;
}

i := 0;
while(i < b1_i)
  invariant(i >= 0 && i <= b1_i);
  invariant(forall e:int :: (e >= 0 && e < b0_i) ==>
      (b0[e] == a[permA[e]]));
  invariant(forall e:int :: (e >= 0 && e < i) ==> (
     b1[e] == a[permA[e + b0_i]]));
  invariant (forall k: int :: 0 <= k && k < b0_i ==>
      lo <= permA[k] && permA[k] <= hi);
  invariant (forall k: int :: b0_i <= k && k < i+
     b0_i ==> lo <= permA[k] && permA[k] <= hi);

  invariant(forall f:int :: (f >= b0_i && f < b0_i +
      i ==> permA[f] == permAtoB1[f-b0_i]));

  invariant(forall e:int :: (e >= 0 && e < b0_i + i
     ==> (permA[e] == permAtoB0[e] || permA[e] ==
     permAtoB1[e-b0_i])));
  invariant (forall e,f:int :: (e >= 0 && e < i+
     b0_i && e != f && f >= 0 && f < i + b0_i ==>
     permA[e] != permA[f]));
{
  permA[i+b0_i] := permAtoB1[i];
  i := i + 1;
}

i := 0;
while(i < b2_i)
  invariant(i >= 0 && i <= b2_i);
```

```
  invariant(forall e:int :: (e >= 0 && e < b0_i) ==>
      (b0[e] == a[permA[e]]));
  invariant(forall e:int :: (e >= 0 && e < b1_i) ==>
      (b1[e] == a[permA[e + b0_i]]));
  invariant(forall e:int :: (e >= 0 && e < i) ==> (
     b2[e] == a[permA[e + b0_i + b1_i]]));
  invariant (forall k: int :: 0 <= k && k < b0_i ==>
      lo <= permA[k] && permA[k] <= hi);
  invariant (forall k: int :: b0_i <= k && k < b1_i+
     b0_i ==> lo <= permA[k] && permA[k] <= hi);
  invariant (forall k: int :: b0_i + b1_i <= k && k
     < b0_i + b1_i + i ==> lo <= permA[k] && permA[k
     ] <= hi);

  invariant(forall f:int :: (f >= b0_i+b1_i && f <
     b0_i + b1_i + i ==> permA[f] == permAtoB2[f-
     b0_i-b1_i]));
  invariant(forall e:int :: (e >= 0 && e < b0_i +
     b1_i + i ==> (permA[e] == permAtoB0[e] || permA
     [e] == permAtoB1[e-b0_i] || permA[e] ==
     permAtoB2[e-b1_i-b0_i])));

  invariant (forall e,f:int :: (e >= 0 && e < b1_i+
     b0_i && e != f && f >= 0 && f < b1_i + b0_i ==>
      permA[e] != permA[f]));
  invariant (forall e,f:int :: (e >= 0 && e < i+
     b1_i + b0_i && e != f && f >= 0 && f < i + b1_i
      + b0_i ==> permA[e] != permA[f]));
{
  permA[i + b0_i + b1_i] := permAtoB2[i];
  i := i + 1;
}

// index mapping from bucket indexes to array
   indexes
off_b0 := lo + b0_i; // end index of bucket0 in a (
   exclusive)
off_b1 := lo + b0_i + b1_i; // end index of bucket1
   in a (exclusive)
off_b2 := lo + b0_i + b1_i + b2_i; // end index of
   bucket2 in a (exclusive)

// sort all buckets
if(b0_i > 0) { call b0_sorted, perm0 := quickSort(b0
```

```
    , 0, b0_i-1); }
  if(b1_i > 0) { call b1_sorted, perm1 := quickSort(b1
    , 0, b1_i-1); }
  if(b2_i > 0) { call b2_sorted, perm2 := quickSort(b2
    , 0, b2_i-1); }

  i := lo;
  while(i < off_b0)
    // bucket values are copied to a
    invariant (forall k: int :: lo <= k && k < i ==> a
      [k] == b0_sorted[k-lo]);
    invariant (forall k: int :: lo <= k && k < i ==> a
      [k] == old(a[perm[k]]));
    invariant (forall k: int :: lo <= k && k < i ==>
      lo <= perm[k] && perm[k] <= hi);

    invariant(forall f:int ::(f >= lo && f < i ==>
      perm[f] == permA[perm0[f-lo]]));
    invariant(forall e,f:int :: (e >= lo && e < i && e
       != f && f >= lo && f < i ==> perm[e] != perm[f
      ]));
  {
    a[i] := b0_sorted[i-lo];
    perm[i] := permA[perm0[i-lo]];
    i := i + 1;
  }

  // copy back elements from bucket to array
  i := off_b0;
  while(i < off_b1)
    // needed to ensure that only this part of array
      is modified
    invariant off_b0 <= i && i <= off_b1;
    // previous knowledge about a is preserved
    invariant (forall k: int :: lo <= k && k < off_b0
      ==> a[k] == b0_sorted[k-lo]);
    // bucket values are copied to a
    invariant (forall k: int :: off_b0 <= k && k < i
      ==> a[k] == b1_sorted[k-off_b0]);

    //permutation
    invariant (forall k: int :: lo <= k && k < off_b0
      ==> a[k] == old(a[perm[k]]));
    invariant (forall k: int :: off_b0 <= k && k < i
```

```
    ==> a[k] == old(a[perm[k]]));
  invariant (forall k: int :: lo <= k && k < off_b0
    ==> lo <= perm[k] && perm[k] <= hi);
  invariant (forall k: int :: off_b0 <= k && k < i
    ==> lo <= perm[k] && perm[k] <= hi);

  invariant(forall f:int ::(f >= off_b0 && f < i ==>
     perm[f] == permA[perm1[f-b0_i-lo] + b0_i]));
  invariant(forall e,f:int :: (e >= lo && e < i && e
     != f && f >= lo && f < i ==> perm[e] != perm[f
    ]));
{
  a[i] := b1_sorted[i-off_b0];
  perm[i] := permA[perm1[i-b0_i-lo] + b0_i];
  i := i + 1;
}

i := off_b1;
while(i < off_b2)
  // needed to ensure that only this part of array
     is modified
  invariant off_b1 <= i && i <= off_b2;
  // previous knowledge about a is preserved
  invariant (forall k: int :: lo <= k && k < off_b0
    ==> a[k] == b0_sorted[k-lo]);
  invariant (forall k: int :: off_b0 <= k && k <
     off_b1 ==> a[k] == b1_sorted[k-off_b0]);
  // bucket values are copied to a
  invariant (forall k: int :: off_b1 <= k && k < i
    ==> a[k] == b2_sorted[k-off_b1]);

  //permutation
  invariant (forall k: int :: lo <= k && k < off_b0
    ==> a[k] == old(a[perm[k]]));
  invariant (forall k: int :: off_b0 <= k && k <
     off_b1 ==> a[k] == old(a[perm[k]]));
  invariant (forall k: int :: off_b1 <= k && k < i
    ==> a[k] == old(a[perm[k]]));
  invariant (forall k: int :: lo <= k && k < off_b0
    ==> lo <= perm[k] && perm[k] <= hi);
  invariant (forall k: int :: off_b0 <= k && k <
     off_b1 ==> lo <= perm[k] && perm[k] <= hi);
  invariant (forall k: int :: off_b1 <= k && k < i
    ==> lo <= perm[k] && perm[k] <= hi);
```

```
    invariant(forall f:int ::(f >= off_b1 && f < i ==>
        perm[f] == permA[perm2[f-b0_i-b1_i-lo] + b0_i
      + b1_i]));
    invariant(forall e,f:int :: (e >= lo && e < i && e
        != f && f >= lo && f < i ==> perm[e] != perm[f
      ]));
  {
    a[i] := b2_sorted[i-off_b1];
    perm[i] := permA[perm2[i-b0_i-b1_i-lo] + b0_i +
      b1_i];
    i := i + 1;
  }
}
```

Appendices

A Provided Code

A.1 Unadulterated Eiffel Code

```
class
        SV_AUTOPROOF

feature
        lst: SIMPLE_ARRAY [INTEGER]
```

A.1.1 wipe

```
feature
wipe (x: SIMPLE_ARRAY [INTEGER])
note
        explicit: wrapping
require
        x /= Void
        modify (x)
local
        k: INTEGER
do
        from
                k := 1
        invariant
                x.is_wrapped
                x.count = x.count.old_
                -- ADD MISSING LOOP INVARIANT(S)
        until
                k > x.count
        loop
                x [k] := 0
                k := k + 1
        end
ensure
        x.count = old x.count
        across 1 |..| x.count as i all x.sequence [i.
           item] = 0 end
end
```

A.1.2 mod_three

```
mod_three (a, b: SIMPLE_ARRAY [INTEGER])
note
        explicit: wrapping
require
        a /= Void
        b /= Void
        a /= b
        a.count = b.count
        a.count > 0
        modify (a, b)
local
        k: INTEGER
do
        wipe (a)
        wipe (b)
        from
                k := 1
        invariant
                a.is_wrapped and b.is_wrapped
                a.count = a.count.old_
                -- ADD MISSING LOOP INVARIANT(S)
        until
                k > a.count
        loop
                if k \\ 3 = 0 then
                        b [k] := a[k] + 1
                end
                k := k + 1
        end
ensure
        across 1 |..| b.count as i all (i.item \\ 3 =
           0) implies b.sequence [i.item] = 1 end
end
```

A.1.3 swap

```
feature
swap (x, y: INTEGER)
note
        explicit: wrapping
require
        lst.is_wrapped
        1 <= x and x <= lst.count
        1 <= y and y <= lst.count
        modify (lst)
local
        z: INTEGER
do
        z := lst [x]
        lst [x] := lst [y]
        lst [y] := z
ensure
        lst.is_wrapped
        lst.count = old lst.count
        across 1 |..| lst.count as i all i.item /= x
           and i.item /= y implies lst.sequence [i.
           item] = (old lst.sequence) [i.item] end
        lst.sequence [x] = (old lst.sequence) [y]
        lst.sequence [y] = (old lst.sequence) [x]
end
```

A.1.4 swapper

```
swapper
note
        explicit: wrapping
require
        lst.is_wrapped
        lst /= Void
        modify (lst)
local
        x, y: INTEGER
do
        from
                x := 1
                y := lst.count
        invariant
                lst.is_wrapped
                lst.sequence.count = lst.sequence.old_
                    .count
                -- ADD MISSING LOOP INVARIANT(S)
        until
                y <= x
        loop
                swap (x, y)
                x := x + 1
                y := y - 1
        end
ensure
        lst.is_wrapped
        lst.sequence.count = (old lst.sequence).count
        across 1 |..| lst.count as i all lst.sequence
            [i.item] = (old lst.sequence) [lst.count -
            i.item + 1] end
end
```

A.1.5 search

```
feature
search (v: INTEGER): BOOLEAN
note
        status: impure
require
        lst.is_wrapped
        lst /= Void
local
        k: INTEGER
do
        from
                k := lst.count
                Result := False
        invariant
                -- ADD MISSING LOOP INVARIANT(S)
        until
                Result or k < 1
        loop
                if lst [k] = v then
                        Result := True
                else
                        k := k - 1
                end
        variant
                k - if Result then 1 else 0 end
        end
ensure
        -- ADD MISSING POSTCONDITION(S)
end
```

A.1.6 prod_sum

```
feature
xx, zz: INTEGER

prod_sum (y: INTEGER)
require
        xx >= 0
        zz >= 0
        y > 0
do
        from
                zz := 0
        invariant
                is_open
                inv
                -- ADD MISSING LOOP INVARIANT(S)
        until
                xx < y
        loop
                zz := zz + 1
                xx := xx - y
        end
ensure
        zz * y + xx = old xx
end
```

A.1.7 paly

```
feature
paly (a: SIMPLE_ARRAY [INTEGER]): BOOLEAN
note
        explicit: wrapping
require
        a /= Void
local
        x, y: INTEGER
do
        from
                x := 1
                y := a.count
                Result := True
        invariant
                -- ADD MISSING LOOP INVARIANT(S)
        until
                x >= y or not Result
        loop
                if a [x] /= a [y] then
                        Result := False
                end
                x := x + 1
                y := y - 1
        variant
                y - x
        end
ensure
        -- ADD MISSING POSTCONDITION(S)
end

end --end of class
```

A.2 Boogie Template

```
// Introduce a constant 'N' and postulate that it is
   non-negative
const N: int;
axiom 0 <= N;

// Declare a map from integers to integers.
// 'a' should be treated as an array of 'N' elements,
   indexed from 0 to N-1
var a: [int]int;

// Returns true iff the elements of 'arr' are small (i
   .e. values in the range -3N to +3N)
function has_small_elements(arr: [int]int): bool
{
  (forall i: int :: (0 <= i && i < N) ==> (-3 * N <=
     arr[i] && arr[i] <= 3 * N))
}

// Sorts 'a' using bucket sort or quick sort, as
   determined by has_small_elements(a)
procedure sort() returns ()
  modifies a;
{
  if (has_small_elements(a))
  {
      // sort 'a' using bucket sort
  } else
  {
      // sort 'a' using quick sort
  }
}
```