Software Verification – Course Project Report

Benjamin Weber
ETH ID 11-933-017
benweber@student.ethz.ch

Marcel Mohler
ETH ID 09-922-998
mohlerm@student.ethz.ch

ABSTRACT

By using *Software Verification* a programmer wants to achieve guarantees in terms of previously defined properties.
However, generally it is undecidable whether a given piece of code conforms to its specification, hence researchers have proposed and developed various semi-automatic methods.
One approach called *Auto-Active Verification* tries to automate as much as possible, while the programmer provides guidance indirectly through program-level annotations[10][4]. We used the verifier AutoProof[1] to verify different algorithms on the program level.
Furthermore, we present an implementation of quicksort[6] and bucketsort[5] in the *intermediate verification language* (IVL) Boogie[8][3]. We provide the needed annotations to formally prove correctness of the implementation with respect to the specification.

1. INTRODUCTION

As part of the **software verification** course offered by **ETH Zurich**, we had to complete a project involving software verification on the program level with AutoProof as well as on the intermediate level through Boogie.
In Section 2 we discuss software verification using AutoProof. We continue to discuss the Boogie part in Section 3 and conclude with some final words in Section 4.

2. AUTOPROOF VERIFICATION

Many features had a specification attached already. Thus, it was sufficient to make the AutoProof tool accept the implementation using annotations such as assertions or invariants. Features with incomplete specifications were more challenging as they did not allow for such a high certainty regarding the correctness of the solution.

2.1 Verification of SV_AUTOPROOF

The **wipe** feature could successfully be verified once we specified in the loop invariant that the postcondition was gradually established. This strategy turned out to be useful for the whole project and both verifiers.

The **mod_three** feature could be verified similarly. We just had to specify what did not get modified additionally.

We needed to specify more to verify the **swapper** feature. To establish the preconditions of **swap** two invariants regarding **x** and **y** were necessary. Once those preconditions could be established, the previous strategy was sufficient to verify the whole feature.

The **search** feature is special because we also had to specify the postconditions ourselves. No tool can verify whether the postconditions truly capture the full extent of what the programmer means to achieve.
We believe the correct meaning of the **search** feature is that it does not modify the list to be searched and that it returns **true** if and only if the given integer is in that list.
The **has** feature of the ghost-feature **sequence** was very useful to express the intuition above in AutoProof.
To verify the feature it was sufficient to explicitly state the implications of **Result** for both cases **true** and **false**.

Verifying the **prod_sum** feature was rather trivial by specifying that the postcondition was gradually established.

In contrast the **paly** feature was more complicated. First some preconditions had to be established by adding simple invariants about **x** and **y**. Then we used the strategy to show AutoProof that the postcondition is gradually established by the loop.
But that still was not enough. Thus we had to specify the counter-example when **Result** became **false** explicitly.
Again, we had to be careful that the postcondition fully captured the meaning of the **paly** feature. We believe we achieved this by establishing the correctness through the postcondition involving the result and by establishing that the parameter does not get modified.

2.2 Experience using AutoProof

It is noteworthy that AutoProof's modify clause is very useful for specification. Because every specification should technically contain what does not get modified in addition to describing what gets modified. Due to the modify clause this becomes much easier, because AutoProof can simply assume that everything not part of the modify clause stays unmodified. In particular, this is something Boogie does not offer and the effects are described in Section 3.2.

Furthermore, the ghost features of **SIMPLE_ARRAY** were very useful to reason about sequences and it seems that AutoProof can handle them pretty well.

3. BOOGIE

Boogie is an intermediate verification language (IVL), intended as a layer on which to build program verifiers for other languages[2].
It is also the name of a tool that accepts Boogie language as input and then generates verification conditions that are passed to an SMT solver such as Z3[9][7].
Initially, it has been developed by Microsoft Research but it is also open to other contributors on GitHub now. In this project we were using both, the tool and the language itself but because of the tight coupling we will treat them synonimously.

3.1 Boogie implementation

Using Boogie to implement the given algorithms did not pose a challenge. Even though Boogie's simplicity, both algorithms could be expressed well. While Boogie has powerful means to abstract complicated concepts[**?**], we only needed some very simple language structures of Boogie to

express an array to be sorted.
One particularly interesting design choice was to implement the partition step of the **quicksort** algorithms in such a way that it could be reused by the **bucketsort** algorithm.
This simplified the implementation due to the additional code reuse and did not make the verification or specification more complicated.

3.2 Boogie specification

In principle, our sort implementations have to fulfill two properties.
Most importantly, the resulting array has to be sorted. This is pretty trivial and we used a simple Boolean function to express this specification.

```
function sorted(a: [int]int, lo, hi: int): bool
{
  hi - lo >= 1
  &&
  (forall i, j: int :: lo <= i
  && i <= j
  && j <= hi ==> a[i] <= a[j]
  )
}
```

Furthermore we need to make sure, that the sorted array is actually a valid permutation of the input array. Modeling this property turned out to be the most difficult part of this project.
The art is to find a way to express the permutation property in such a way that Boogie can handle it well and that it is also useful to work with for verification and specification.
As such we tried several different solutions which we will describe in more detail in the following.

A very early idea was to define the properties with axioms.

```
// uninterpreted function describing permutations
function permutation(a, b: [int]int): bool;

// reflexivity
axiom (forall a: [int]int :: permutation(a, a));
// symmetry
axiom (forall a, b: [int]int :: permutation(a, b)
  ==> permutation(b, a));
// transitivity
axiom (forall a, b, c: [int]int ::
  permutation(a, b) && permutation(b, c)
    ==> permutation(a, c));
```

This allows one to easily introduce trivial properties of permutations such as reflexivity, symmetry and transitivity. Once, one can prove that swap preserves the permutation property, Boogie can infer that whole procedures also preserve it, if they only use the swap procedure to modify the input array (which can be easily achieved).
While this is an attractive approach as most of the complexity is contained in the swap procedure, it is also dangerous in general. The problem with axioms and assumptions is that Boogie will assume them to be true unconditionally. Therefore, one has to be very careful not to introduce unsound axioms or assumptions.
In a sense, one loses some of the certainty gained by using sound tools such as Boogie, because correctness additionally relies on the correctness and soundness of the axioms and assumptions.
In this specific case it may be reasonable to use such trivial axioms, however we did not like this solution.

A more interesting approach is to use Boogie to prove these axioms automatically. Using our definition of **perm_of** one could use only the following axioms to prove these properties of permutations:

```
axiom (forall a, b: [int]int :: permutation(a, b)
   ==> (exists m: [int]int :: perm_of(a, b, m)));
axiom (forall a, b, m: [int]int :: perm_of(a, b, m)
   ==> permutation(a, b));
```

And then one could use a procedure to give a constructive proof for the reflexivity of permutations:

```
procedure reflexivity(a: [int]int)
  ensures permutation(a, a);
{
   var m: [int]int;
   var k: int;

   k := 0;
   while (k < N)
      invariant 0 <= k && k <= N;
      invariant (forall i: int :: 0 <= i && i < k
          ==> m[i] == i);
   {
      m[k] := k;
      k := k + 1;
   }
   assert perm_of(a, a, m);
}
```

Then this gives strong reason to the legitimacy of using the reflexivity axiom during verification. For symmetry one would have to make the following procedure verify:

```
procedure symmetry(a, b: [int]int)
   requires permutation(a, b);
   ensures permutation(b, a);
{
   var m, _m: [int]int;
   // this is allowed due to the axioms
   // linking permutation and perm_of
   assume perm_of(a, b, m);

   /* constructive proof */

   assert perm_of(b, a, _m);
}
```

And for transitivity one could use a similar approach as used for symmetry.
However, we were not able to verify the symmetry and transitivity property this way. Furthermore, Boogie might not be the right tool for such an approach and there exist other ones that excel at these disciplines. Nevertheless, it was an interesting excursion into another aspect of verification.

Another idea which we explored was to look at the histograms of two arrays to check whether they were permutations of each other. One can (recursively) define a Boogie function **count** which determines how often a certain element appears in an array. If every element of one array has the same count in another array, then they are permutations of each other.
This has the advantage that no permutation array is required for verification.
However, it turned out that Boogie can not handle these counts well and we were unable to verify the permutation property with this approach.

Because of the mentioned problems and pitfalls of the previous attempts we decided to model the property by keeping an additional permutation array, that stores the modification from the input array to the final, sorted array. This allows us to easily verify that the current array is a permutation of the initial one. Further details can be found in Section 3.3.

In general, expressing the specifications on Boogie was interesting due to two different aspects.

Firstly, there was the challenge to fully capture the meaning of an algorithm through the specification.
Boogie was very helpful there with its rather powerful expression language which contains both the universal and existential quantifiers. However, one also had to be careful to express formulas such that Boogie could handle them well. Otherwise, more annotations are necessary which is less desirable.

Secondly, it was often surprising to see what a complete specification was for a procedure. An incomplete specification can lead to the failure of the verification. Thus we often came across incomplete specifications.
We learned that in addition to specifying what did get changed we also had to specify what did not get changed. A complicated example of this can be found in the **quicksort** and **partition** procedures.
Both of these procedures preserve the property that if some values were previously bigger-equal than all values in a given range, those values will still be bigger-equal after the procedure call.
While this is trivially obvious because both procedures only permute an array, it is not obvious that this property is necessary to verify the **quicksort** and **bucketsort** implementations.

3.3 Boogie verification

We were able to verify the correctness of our implementation based on the sorted and permutation properties.
We will describe a few challenges and issues we encountered during the verification process in the following.

One of the main issues we had was termination. In Boogie one can only prove partial correctness, i.e. correctness only for the program traces that do terminate. Boogie ignores program traces that do not terminate. It would have been very useful to be able to prove total correctness which means that the program is guaranteed to terminate for all input parameters in addition conforming to its specification.
There is the concepts of a loop variants (a condition that decreases on each loop iteration) that extend the axiomatic semantics with total correctness but Boogie does not support this.
Applied to our scenario this meant that if we do not carefully design our algorithms such that they always terminate, we might end up with postconditions that evaluate to trivially true because they can never be reached.
See this small example, which verifies correctly in Boogie even though `1 == 0` is clearly a false statement.

```
1 procedure F(n: int) returns (r: int)
2   ensures 1 == 0;
3 {
4   while(true) {
5     r := n;
6   }
7   r := 0;
8 }
```

If we change the `while(true)` in line 4 to `while(false)` Boogie correctly reports that the postcondition might not hold.

We also found the fact that we can not define preconditions for functions to be rather limiting. Taking the example of `sorted()`, we would have liked to require $N \geq 2$ because sortedness does not make sense for lists of length 1 or less. In our implementation we decided to conservatively define **sorted** to be false for those cases. However, this definition can still yield false positives if the expression **!sorted(...)** is used. A precondition would have allowed us to handle this better. This is similar to division which is also undefined for divisor being zero.
Additionally, this forces one to think of the cases where $N < 2$ and it made us realize that we should specify the behavior of our implementation also for this case. We believe in this case a sorting function should leave the array untouched. If one does not specify this, a sorting function could be free to do anything with the input array if its length is 1 or less!

Another observation is the difference between defining a pre- and postcondition and defining a single postcondition which includes an implication. An example is drawn below:

```
procedure quickSort_a(lo, hi: int)
  ensures perm_of(old(a), old(old_a), old(perm))
  ==> perm_of(a, old_a, perm);
{..}
procedure quickSort_b(lo, hi: int)
  requires perm_of(a, old_a, perm);
  ensures perm_of(a, old_a, perm);
{..}
```

Option a is better (and we used this in our implementation) because preconditions should be requirements of the algorithm to work correctly. However, **quicksort** works correctly without **perm_of(a, old_a, perm)**.
This way the algorithm can be used without having to establish the permutation property and instead it can be voluntarily established, if one wishes to make use of it.

One interesting thing we noticed with preconditions is that they lead to ad-hoc code. For example the precondition $lo <= hi$ of the **quicksort** procedure leads to an additional **if** clause for every single call of **quicksort**.
The only exception being in the **sort** procedure, because it implicitly establishes this precondition. We are unsure whether this side-effect was introduced deliberately or not. The following assertion can be verified by Boogie:

```
assert !has_small_elements(a) ==> N > 0;
```

However, with regards to the ad-hoc code due to preconditions, one might consider to remove these preconditions and simply deal with those cases separately inside the procedure. Though, we believe this solution is not optimal, because one can not be sure that the procedure handles these cases correctly in all situations.
Thus, we see this ad-hoc code as a cost, to ensure that these cases are handled correctly (in our case it is just a coincidence that when $lo > hi$ **quicksort** should simply do nothing for every **quicksort** call).

To understand the similarities and differences between invariants and pre-/postconditions we found Section 9.8 of the Boogie language manual[**?**] very helpful and worth mentioning. A pre-/postcondition pair can basically be seen as a contract between the implementation and the call site. Preconditions are assumed by the implementation and checked at the call site. On the other hand, a loop invariant is also a contract between two parties namely the iterations of the past and the iterations of the future. And similarly, the past checks the invariant and the future assumes it.
However, there seem to be a difference between using invariants or simply working with asserts (to model pre- and postconditions of loops). In the following code example F verifies correctly, while G does not.

```
const N : int;
axiom N >= 0;
procedure F() returns (k : int)
{
  k := 0;
  while(k < N)
  invariant k <= N;
  {
    k := k+1;
  }
  assert k == N;
}

procedure G() returns (k : int)
{
  k := 0;
  assert k <= N;
  while(k < N)
  {
    k := k+1;
21       assert k <= N;
22     }
23     assert k == N;
24   }
```

This is unfortunate, as G should also be correct in terms of Hoare Logic[**?**]. Nevertheless, Boogie seems to not transfer the result of the assert in line 21 outside of the loop. We do not think that this is a problem since it does not decrease the expressivness of Boogie but an interesting fact.

Please note, that we also carefully annotated the attached code with many comments and explanations which exceed the scope of this report.

4. CONCLUSION

We found it very exciting to apply the concepts of software verification to larger examples. In the lectures we have always seen these concepts being applied to small and simple functions, where verification was rather trivial. And while **quicksort** and **bucketsort** are still only a few dozen lines of code, verification turned out to be a lot more complicated than expected.
Though, we also gained a much deeper understanding of these algorithms, particularly when it comes to which properties allow us to use them in a modular way.
We appreciated working both on the program level as well as on the intermediate verification language level to get a bigger picture of the whole software verification process.
With AutoProof building on top of Boogie, we could reason about the program on a higher level. Furthermore, AutoProof already provides some higher level structures such as the **sequence** ghost-feature as well as others, which are very useful to reason about a program.
In contrast Boogie gives the user a bigger burden to create helpful higher-level structures themselves. It was interesting to see how different choices regarding this affected verification. With this burden comes also the advantage that one has the possibility to choose using certain structures such that verification can be greatly simplified.
In contrast, AutoProof's abstractions are set and the user loses freedom in this aspect. Nevertheless, this also comes with the advantage that tools such as AutoProof can expose specialized abstractions to the user. Abstractions that are tailored to programming languages such as Eiffel.
In conclusion, we found this project to be quite rewarding with respect to lessons learned and we believe it brought us forward as software engineers. We would recommend such software verification tools to others and welcome using them ourselves for future projects.

5. REFERENCES

[1] AutoProof Website. `http://se.inf.ethz.ch/research/autoproof/`. Accessed on 19 Nov 2015.

[2] Boogie Github. `https://github.com/boogie-org/boogie`. Accessed on 26 Nov 2015.

[3] Microsoft Research: Boogie website. `http://research.microsoft.com/en-us/projects/boogie/`. Accessed on 19 Nov 2015.

[4] Software Verfification Lecture ETH Zurich. `http://se.inf.ethz.ch/courses/2015b_fall/sv/slides/05-AutoActiveVerification.pdf`. Accessed on 25 Nov 2015.

[5] Wikipedia: Bucket sort. `https://en.wikipedia.org/wiki/Bucket_sort`. Accessed on 25 Nov 2015.

[6] Wikipedia: Quick sort. `https://en.wikipedia.org/wiki/Quicksort`. Accessed on 25 Nov 2015.

[7] Z3 SMT Solver Github. `https://github.com/Z3Prover/z3`. Accessed on 26 Nov 2015.

[8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2006.

[9] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[10] J. Tschannen, C. A. Furia, M. Nordio, and B. Meyer. Verifying eiffel programs with boogie. *arXiv preprint arXiv:1106.4700*, 2011.