Problem Sheet 4: Boogie and Boogaloo

Chris Poskitt and Nadia Polikarpova
ETH Zürich

Starred exercises (∗) are more challenging than the others.

1 Introduction

In the problem class two weeks ago, we worked with the AutoProof tool, a static verifier for Eiffel programs. Recall, as illustrated in Figure 1, that the tool performs this verification by *translating* Eiffel programs—and their contracts (i.e. their specifications)—into an *intermediate verification language* called Boogie. The Boogie verifier then performs a further translation into *verification conditions*; logical formulae which, if valid, imply the correctness of your original Eiffel program with respect to its contracts. An SMT solver is then used to try and automatically determine whether or not these formulae are valid, and hence, whether or not the original program is correct.

While hopefully you were able to verify a number of Eiffel classes, you may have hit some of the following problems: a lack of expressiveness in the specification language, a lack of concrete counterexamples in failing verification attempts, or a lack of guidance when a "correct" implementation cannot be verified because its specification is insufficient. In this problem class, we return to the issue of automatic program verification, but from a different angle; one that helps alleviate these concerns.

2 Boogie and Boogaloo

Firstly, rather than interacting with Boogie via AutoProof, now we will work with the Boogie verifier directly, writing programs and specifications directly in its front-end language [1]. Boogie provides a much richer specification and implementation language, including among its features: nondeterminism, unbounded quantification, and infinite structures (e.g. sets). For an introduction to the language and verifier, consult the following slides and manual:

- Nadia Polikarpova's Boogie/Boogaloo slides:
  `http://se.inf.ethz.ch/courses/2013b_fall/sv/slides/G3-Boogie-Boogaloo.pdf`
- Boogie manual:
  `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`

Boogie, unsurprisingly, suffers from two of the same problems as AutoProof: an absence of counterexamples when implementations do not agree with specifications, and little means of validation when the implementation is actually correct but the specification insufficient for the proof to go through automatically. Hence we will also be using the Boogaloo tool, which provides a means to automatically generate executions of Boogie programs, in order to better understand why a verification attempt is failing.

Boogaloo [2] is an interpreter and run-time assertion checker for Boogie. Based on a technique called *symbolic execution*, it enumerates all the possible paths of a Boogie program (there

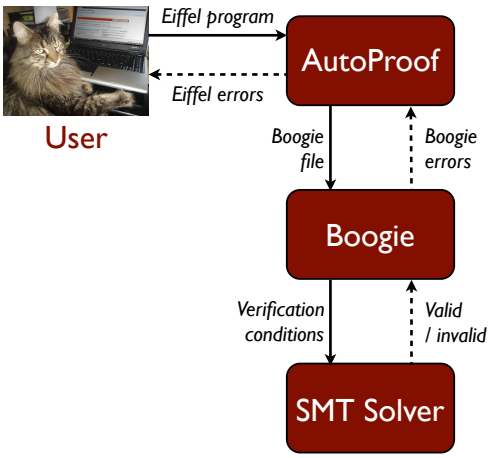

Figure 1: The AutoProof workflow

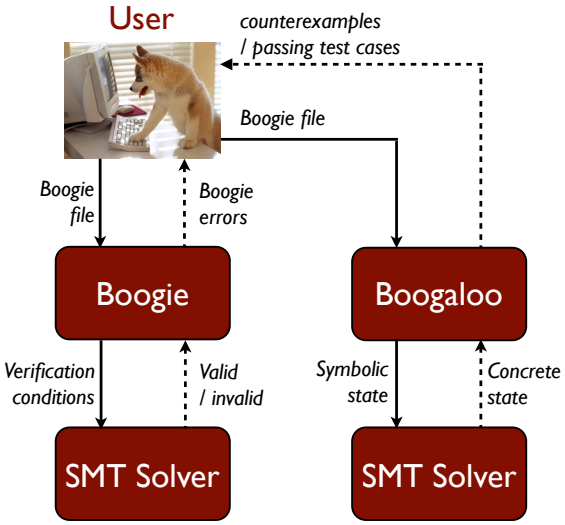

Figure 2: The Boogie/Boogaloo workflow

may be several due to the nondeterminism), but does so without having to enumerate all the possible inputs too: instead, it uses *symbolic values*—essentially "placeholders" for concrete program values. Upon arriving at the end of a path, Boogaloo will have a constraint built up of such symbols to solve; each solution representing concrete values that variables could *possibly* have in this execution. Such solutions are reported to the user, providing either (1) concrete counterexamples for understanding why a verification attempt failed; or (2) passing test cases, providing some validation of the implementation. This workflow is summarised in Figure 2.

For an introduction to Boogaloo, and an informative, concise manual, please consult:

- Nadia Polikarpova's Boogie/Boogaloo slides:
  `http://se.inf.ethz.ch/courses/2013b_fall/sv/slides/G3-Boogie-Boogaloo.pdf`
- Boogaloo wiki and manual:
  `https://bitbucket.org/nadiapolikarpova/boogaloo/wiki/Home`

3 Setting Up

Both Boogie and Boogaloo are available to try in your browser, through rise4fun and Comcom respectively:

`http://rise4fun.com/boogie`

`http://cloudstudio.ethz.ch/comcom/#Boogaloo`

Should you prefer, both tools can also be installed on your laptop. Please refer to their respective websites for details:

`http://research.microsoft.com/en-us/projects/boogie/`

`https://bitbucket.org/nadiapolikarpova/boogaloo/wiki/Home`

4 Exercises

i. The Boogie program `Fibonacci` in Figure 3 is buggy: use Boogaloo to debug it, verify it in Boogie, and then return to Boogaloo to generate some valid executions.

Hint: in the command box on Comcom, type `test` for test mode, `-p PROC` to specify a procedure under test, and `-o N` to generate N valid executions.

ii. Prove the following specification in Boogie:

$$\{\text{true}\}\ t := x;\ x := x + y;\ y := t\ \{x = x' + y' \wedge y = x'\}$$

where x', y' respectively denote the values of x and y in the pre-state.

iii. Consider the Boogie program `ArraySum` in Figure 4 which is supposed to recursively compute the sum of array elements. Try to debug the program using Boogaloo and then verify it in Boogie.

Hint: don't forget about loop invariants! Without an invariant, any loop in Boogie is treated as equivalent to assigning arbitrary values to program variables.

iv. Implement, test, and verify the algorithm `FindZero` (its signature is given in Figure 5), that linearly searches an array for the element 0:

> *Input:* an integer array a, and its length N.
>
> *Output:* an index $k \in \{0, \ldots, N-1\}$ into the array a such that $a[k] = 0$; otherwise $k = -1$.

The specification should guarantee that if there exists an array element $a[i] = 0$ with $0 \leq i < N$, then `FindZero` will always return a k such that $k \geq 0$ and $a[k] = 0$.

v. (∗) Copy the procedure `FindZero` you wrote in part (iv), rename it to `FindZeroPro`, and add the following two preconditions:

```
requires (forall i: int :: 0 <= i && i < N ==> 0 <= a[i]);
requires (forall i: int :: 0 <= i-1 && i < N ==> a[i-1]-1 <= a[i]);
```

These additional preconditions require that along the array, values never decrease by more than one. Adapt your linear search algorithm such that after an iteration of its loop, instead of incrementing the current index k by 1, it now increments it by $a[k]$. Verify that the procedure still establishes the same postconditions as in (iv).

Hint: you will need to prove that that all array values between $a[k]$ and $a[k + a[k]]$ are non-zero (i.e. that 0-values are not skipped over by the search) and use this property in the loop. For this you will need to write more than simply a loop invariant, e.g. some "ghost" (or "proof") code.

vi. (∗) Take a look at the Boogie program `BinarySearch` in Figure 6 which is supposed to perform a binary search[1]. Debug the implementation and add the missing loop invariants with the assistance of Boogaloo.

[1] See: http://en.wikipedia.org/wiki/Binary_search_algorithm

References

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

[2] Nadia Polikarpova, Carlo A. Furia, and Scott West. To run what no one has run before: Executing an intermediate verication language. In *Proc. 4th International Conference on Runtime Verification (RV 2013)*, volume 8174, pages 251–268. Springer, 2013. http://se.inf.ethz.ch/people/polikarpova/publications/rv13.pdf.

Appendix: Code Listings

Hint: the code listings below are all available to download from the course webpage (no need to copy and paste from this PDF!):

http://se.inf.ethz.ch/courses/2013b_fall/sv/

```
/*
  Naive computation of Fibonacci numbers using a recursive procedure.
  An assertion defines the semantics using a recursive function.
*/

// n-th Fibonacci number
function fib(x: int): int;
axiom fib(0) == 0;
axiom fib(1) == 1;
axiom (forall x: int :: x > 1 ==> fib(x) == fib(x - 2) + fib(x - 1));

// Compute n-th Fibonacci number
procedure ComputeFib(x: int) returns (res: int)
  requires x >= 0;
{
  var f1, f2: int;
  if (x > 1) {
    call f1 := ComputeFib(x - 1);
    call f2 := ComputeFib(x - 2);
    res := f1 + f2;
  } else {
    res := 1;
  }
}

// One way to call ComputeFib
procedure Main(x: int) returns (fib: int)
  requires x >= 8;
{
  call fib := ComputeFib(x);
  assert fib == fib(x);
}
```

Figure 3: Program `Fibonacci` (buggy)

```
/*
  Sum of array elements, with the semantics of sum defined recursively.
*/

// Sum of N elements of array a
function recSum(a: [int] int, N: int) returns (int)
{ if N == 0 then 0 else recSum(a, N - 1) + a[N - 1] }

// Iteratively compute the sum of array elements.
procedure Sum(a: [int] int, N: int) returns (sum: int)
  requires 1 <= N;
  ensures recSum(a, N) == sum;
{
  var i: int;
  i, sum := 1, 0;
  while (i < N)
  {
    sum := sum + a[i];
    i := i + 1;
  }
}
```

Figure 4: Program `ArraySum` (buggy)

```
procedure FindZero(a: [int]int, N: int) returns (k: int)
  // specification here
{
  // implementation here
}
```

Figure 5: Program `FindZero` (incomplete)

```
/*
  Binary Search.
*/

// Is array a of length N sorted?
function sorted(a: [int] int, N: int): bool
{ (forall j, k: int :: 0 <= j && j < k && k < N ==> a[j] <= a[k]) }

// Efficiently search for value in a sorted array a of length N.
procedure BinarySearch(a: [int] int, N: int, value: int) returns (index: int)
  requires N >= 0;
  requires sorted(a, N);
  ensures 0 <= index && index <= N;
  ensures index < N ==> a[index] == value;
     // If index is within bounds, value was found
  ensures index == N ==> (forall j: int :: 0 <= j && j < N ==> a[j] != value);
     // If index is out of bounds, value does not occur
{
  var low, high: int;
  low, high := 0, N - 1;
  while (true)
  {
    index := (low + high) div 2;
    assert 0 <= index && index < N; // Language-enforced
    if (value < a[index]) {
      high := index - 1;
    } else {
      low := index + 1;
    }
    if (low > high || value == a[index]) {
      break;
    }
  }
  if (low > high) {
    index := N; // not found
  }
}
```

Figure 6: Program `BinarySearch` (buggy)