Problem Sheet 2: AutoProof

Chris Poskitt and Julian Tschannen
ETH Zürich

"Beware of bugs in the above code; I have only proved it correct, not tried it."
– Donald E. Knuth

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

1 Background

This exercise class is concerned with the AutoProof tool [2, 3], a static verifier for programs written in (a subset of) the object-oriented language Eiffel. The tool takes an Eiffel program—annotated with *contracts* (i.e. executable pre-/postconditions, class invariants, intermediate assertions)—and *automatically* attempts to verify the correctness of the program with respect to its contracts.

The tool is built on top of Boogie [1], an automatic verification framework developed by Microsoft Research. AutoProof translates Eiffel programs and their contracts (i.e. their proof obligations) into the front-end language of Boogie—an *intermediate verification language* encoding the semantics of the source program in terms of primitive constructs, and prescribing what it means for the source program to be correct. The Boogie tool then translates this intermediate program into a set of *verification conditions*; logical formulae which if valid, indicate the correctness of the source program. The validity of these verification conditions is checked automatically by an SMT solver (currently Z3).

This workflow is summarised in Figure 1. We will only be interacting with AutoProof itself in this exercise class, but it is helpful to be roughly aware of how it works and what translations it is performing (in a later class, we will look at the Boogie framework directly).

2 Setting Up

The easiest way to interact with AutoProof is in a web browser, through Comcom:

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

The Eiffel programs from these exercises are provided in Comcom already; simply hit the "Run" button to execute AutoProof . You can verify your own Eiffel programs using the "More AutoProof" tab (note that you cannot save your programs in Comcom, so you should work on them offline and then paste them in).

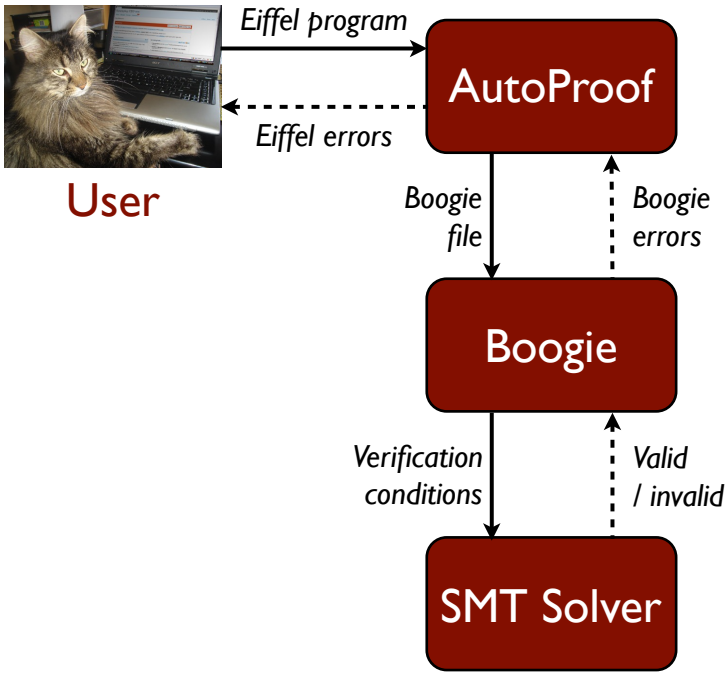

Figure 1: The AutoProof workflow

3 Exercises

i. Consider the class `WRAPPING_COUNTER` in Figure 2. The method `increment` increases by one its integer input; except if the input is 59, in which case it wraps it round to 0. Verify the class in AutoProof *without* changing the implementation, i.e. adding only the necessary preconditions. Strengthen the postcondition further as suggested in the comments, and check that the proof still goes through.

ii. In the axiomatic semantics problem sheet, we encountered several simple program specifications expressed as Hoare triples. Using the class `AXIOMATIC_SEMANTICS` in Figure 3, write some simple contract-equipped methods and show the following in AutoProof:

(A) $\models \{x = 21 \wedge y = 5\}$ `skip` $\{y = 5\}$
(B) $\models \{x > 10\}$ `x` $:=$ `2` $*$ `x` $\{x > 21\}$
(C) $\models \{x \geq 0 \wedge y > 1\}$ `while x < y do x :=` `x` $*$ `x` $\{x \geq y\}$
(D) $\models \{x = 5\}$ `while x > 0 do x :=` `x` $+$ `1` $\{x < 0\}$
(E) $\models \{x = a \wedge y = b\}$ `t :=` `x`; `x :=` `x` $+$ `y`; `y :=` `t` $\{x = a + b \wedge y = a\}$
(F) $\models \{in + m = 250\}$ `while (i > 0) do m :=` `m` $+$ `n`; `i :=` `i` $-$ `1` $\{in + m = 250\}$

Hint: Eiffel does not offer a while construct. Try experimenting with from-until-loop instead, as well as if-then-else with recursion.

iii. Consider the class `MAX_IN_ARRAY` in Figure 4. What does the `max_in_array` method do? Prove the class correct in AutoProof by determining a suitable precondition and loop invariant.

Hint: you might find Eiffel's across-as-all loop construct[1] helpful for expressing loop invariants.

iv. (∗) Consider the class `SUM_AND_MAX` in Figure 5. What does the method `sum_and_max` do? What can you prove about it using AutoProof?

v. (∗∗) Consider the class `LCP` in Figure 6. The method `lcp` implements a Longest Common Prefix (LCP) algorithm[2] with input and output as follows:

> *Input:* an integer array a, and two indices x and y into this array.
>
> *Output:* length of the longest common prefix of the subarrays of a starting at x and y respectively.

What can you prove about the class in AutoProof?

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] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Automatic verification of advanced object-oriented features: The AutoProof approach. In *Tools for Practical Software Verification - LASER 2011, International Summer School*, volume 7682 of *LNCS*, pages 134–156. Springer, 2012. http://se.inf.ethz.ch/people/tschannen/publications/TschannenLASER11.pdf.

[3] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Program checking with less hassle. In *Proceedings of Verified Software: Theories, Tools and Experiments (VSTTE)*. To appear, 2013. http://se.inf.ethz.ch/people/tschannen/publications/tfnm-vstte13.pdf.

[1] See: http://bertrandmeyer.com/2010/01/26/more-expressive-loops-for-eiffel/
[2] From the FM 2012 verification challenge.

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/

and are also all set up in Comcom itself.

```
class WRAPPING_COUNTER

feature
    increment (count: INTEGER)  : INTEGER
    require
        -- preconditions
    do
        if (count = 59) then
            Result := 0
        else
            Result := count + 1
        end
    ensure
      counter_in_range: Result >= 0 and Result < 60

      -- missing postcondition: the method should increment all values
      --   by 1, except those above 58 (which wrap back to 0).
    end
end
```

Figure 2: Class `WRAPPING_COUNTER`

```
class AXIOMATIC_SEMANTICS

feature
    a,b,i,m,n,x,y: INTEGER

feature
    partA
  require
    -- precondition
  do
    -- program
  ensure
    -- postcondition
  end

  -- etc.
end
```

Figure 3: Class `AXIOMATIC_SEMANTICS`

```
class MAX_IN_ARRAY

feature

  max_in_array (a: ARRAY [INTEGER]): INTEGER
      -- Index of maximum element of a.
    note
      pure: True
    require
      -- precondition
    local
      x, y: INTEGER
    do
      from
        x := 1
        y := a.count
      invariant
        -- loop invariant
      until
        x = y
      loop
        if a[x] <= a[y] then
          x := x + 1
        else
          y := y - 1
        end
      variant
        y - x
      end
      Result := x
    ensure
      result_in_range: 1 <= Result and Result <= a.count
      result_is_max: across a as i all i.item <= a[Result] end
    end
end
```

Figure 4: Class MAX_IN_ARRAY

```
class SUM_AND_MAX

feature

  sum_and_max (a: ARRAY [INTEGER]): TUPLE [sum, max: INTEGER]
      -- Calculate sum and maximum of array a.
    note
      framing: False
    require
      -- preconditions
    local
      i: INTEGER
      sum, max: INTEGER
    do
      from
        i := 1
      invariant
        -- loop invariants
      until
        i > a.count
      loop
        sum := sum + a[i]
        if a[i] > max then
          max := a[i]
        end
        i := i + 1
      variant
        a.count - i + 1
      end
      Result := [sum, max]
    ensure
      -- postconditions
    end
end
```

Figure 5: Class SUM_AND_MAX

```
class LCP

feature

  lcp (a: ARRAY [INTEGER]; x, y: INTEGER): INTEGER
    note
      pure: True
    require
      -- preconditions
    do
      from
        Result := 0
      invariant
        -- loop invariants
      until
        x + Result = a.count + 1 or else
        y + Result = a.count + 1 or else
        a[x + Result] /= a[y + Result]
      loop
        Result := Result + 1
      variant
        a.count - Result + 1
      end
    ensure
      -- postconditions
    end
end
```

Figure 6: Class LCP