Independent Study: Tool-Assisted Verification of C Code using Floyd-Hoare Logic

Nikolas Mählmann

April 6, 2020

1 Introduction

This document explains the usage and inner workings of `verify-c`. `verify-c` is a command line tool which verifies programs written in a subset of C. The tool is based on Floyd-Hoare logic, which was intensivley discussed in the course *Korrekte Software: Grundlagen und Methoden* at the University of Bremen held by Serge Autexier and Christoph Lüth in the summer semester 2019. `verify-c` is an educational implementation of the discussed techinques aimed to deepen their understanding and explore which challenges arise when trying to formally verify software. `verify-c` is written in Haskell and the source code is available online at `https://github.com/nmaehlmann/verify-c`.

2 Installation

`verify-c` can be built using the Haskell build tool `stack` by calling:

```
stack install
```

in the root directory. Additionally `verify-c` relies on the `Z3` theorem prover which has to be installed and added to the `PATH` variable. It can be downloaded at `https://github.com/Z3Prover/z3`.

3 Usage

`verify-c` parses program code written in a subset of C. Each function is annotated with logical pre- and postconditions, which specify the contract of the function. Based on the parsed program it generates a set of verification conditions. The verification conditions are exported to a theorem prover, which checks whether or not they are satisfied. If all verification conditions are proven successfully, the implemented functions satisfy their contract.
Lets start with a simple example. Listing 1 shows a verified implementation of the faculty function.

Listing 1: faculty.c0

```
1  int faculty(int n){
2      precondition("n >= 0");
3      postcondition("\result == fac(n)");
4
5      p = 1;
6      c = 1;
7      while(c <= n){
8          invariant("p == fac(c - 1) && c <= n + 1 && c > 0");
9          p = p * c;
10         c = c + 1;
11     }
12     return p;
13 }
```

It is written regular C code but additional function calls have been added to specify the contract of the function in order to verify it. The precondition in l.2 states that the function argument `n` has to be positive. The postcondition in l.3 that after calling the function will return `fac` of `n`. When verified successfully these conditions have the following semantic: If the function `faculty` is called with a positive argument `n` and it terminates, then the return value of this function will equal `fac` of `n`. Furthermore the while loop is annotated with an invariant (l.8). The invariant has to be satisfied before the while loop is entered as well as after each loop is completed. The specification of preconditions, postconditions and invariants is mandatory and missing specifications will result in a parser error.
Additionally to the C source code `verify-c` requires an envrionment file if

custom functions or predicates are used in specifications. The function `fac` used in the precondition and in the invariant is such a custom function. It is specified in Listing 2.

Listing 2: faculty.env

```
(declare-fun fac (Int) Int)
(assert (= (fac 0) 1))
(assert (forall ((nn Int)) (implies (< 0 nn) (= (fac nn) (* nn
    ↪ (fac (- nn 1)))))))
```

The specification of the envrionment is written in the SMT-LIB format and verbatim fed into the `Z3` prover. More information regarding the SMT-LIB language can be found online at `http://smtlib.cs.uiowa.edu/language.shtml`. In order to be found by `verify-c` the environment has to have the same name as the source file but with an `.env` extension. With the environment in place the source code of the faculty function can now be verified by calling:

```
verify-c faculty.c0
```

which produces the following output:

```
Generated 3 verification condition(s). Starting proof:
[1/3] : Precondition faculty : OK
[2/3] : While Case True (1:8) : OK
[3/3] : While Case False (1:8) : OK

Summary: VERIFICATION OK
```

Hooray! Three verification conditions were generated by `verify-c` and successfully proven by `Z3`. One originates from the precondition of the faculty function, two from the invariant of the while loop.
In this case every verification condition could be proven, which is indicated by the status code `OK`. Other status codes are:

- `SIMPLIFY FAILED`: The verification conditions could not be simplified enough to be proven. This is most likely caused by ambiguous dereferencing.
- `SMT EXPORT FAILED`: The verification condition could not be translated into SMT-LIB code. This is most likely caused by ambiguous referencing.

- `VIOLATED`: The verification condition was disproven. The specification and program do not match.
- `TIMEOUT`: `Z3` timed out while trying to prove the verification condition. It could neither disprove nor prove it.
- `SMT ERROR`: `Z3` produced an unkown error.
- `SKIPPED`: The verification condition was skipped because of a previous error.

The generated verification conditions and SMT-LIB code as well as logfiles are stored in the `.\target` folder created by `verify-c`. This is the place to look at in case verification fails.
`verify-c` can be further configured by using command line options. A list of all available can be displayed by calling:

```
verify-c -h
```

which outputs:

```
  Help Options:
  -h, --help
    Show option summary.
  --help-all
    Show all help options.

Application Options:
  --color :: bool
    Whether or not to use ANSI colors.
    default: false
  --timeout :: int
    SMT solver timeout in seconds.
    default: 5
  --no-skip :: bool
    Whether or not to continue verification after a condition
        ↪ could not be
    verified.
    default: false
```

4