Part 2: Implementation of a Simple Program Verifier using Hoare Logic

José Proença & David Pereira & Eduardo Tovar
{pro,drp,emt}@isep.ipp.pt

Formal Verification of Critical Applications – 2021/2022

To do

The objective of this second project of FVOCA is that each group of students completes the implementation of a very simple program verified based on Hoare logic, that uses the Z3 theorem prover to discharge proof obligations.

What to submit

Each group must send, via email to `drp@isep.ipp.pt`, the two Python files that contain the functions that are required to be implemented. They are `WPrec.py` and `VCs.py`.

Require Software

In terms of required software, you need to have any version of Python 3.10 installed in your system (or accessible via the IDE or code editor of your choice). Also, the colorama package is required for pretty printing only. In case you have difficulties installing any of the mentioned software, please contact `drp@isep.ipp.pt` at your earliest convenience.

Deadline

12 June @ 23h59m

1 Objectives

As announced in the classes, the objective for each group is to complete the implementation of two functions that are incomplete in the distributed Python code available at `https://github.com/cister-labs/hoare_project`, and that serves as code base for this project. These incomplete functions are:

1. **wprec:** the function that implements the generation of *weakest preconditions*. Its code is available in the file `WPrec.py`.
2. **VC:** the function that implements the generation of verification conditions. Its code is available in the file `VCs.py`.

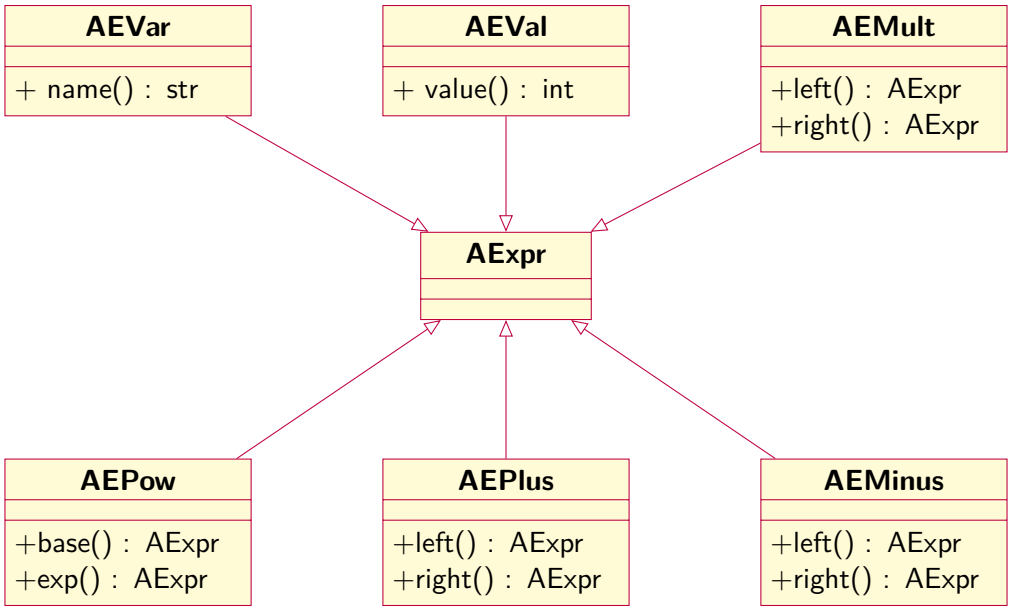

Figure 1: Arithmetic expressions class hierarchy.

To implement these functions, you should follow their algorithmic definitions as presented in the slides used during the classes and also read and understand how the improved verification generation function has been implemented (the function is named **VC_i** and can be found on file `VCs.py`).

The specifications of the two functions for the groups to complete are presented later on on this document. But, before that, we will provide an high-level description of the parts of the code base to ease its understanding.

1.1 Class Hierarchy for Arithmetic and Boolean Expressions, Specifications, and Commands

The implementation that is going to be used considers the simple imperative language introduced in the classes. As we have seen before, the syntax of that language, to which we call IMP, is implemented via a class hierarchy defined in the file `Exprs.py`. The diagram presented in Fig. 1 shows the defined hierarchy for the case of arithmetic expressions.

In terms of usage, if we want, for instance, to represent the arithmetic expression $2 \times (x + y^2)$, we have to write the following Python code:

```
x = AEMult(AEVal(2),AEPlus(AEVar('x'),AEPow(AEvar('y'),AEVal(2))))
```

The class that implements constructs related to multiplication and that used to build the above expression is derived from parent class named AExpr (which is empty and serves just the purpose of allowing to build the class hierarchy for arithmetic expressions).

```
''' Parent class for remaining arithmetic expressions cases'''
class AExpr:
    pass

''' Class that represents the multiplication of two arithmetic expressions.'''
```

```
class AEMult(AExpr):


    def __init__(self,l,r):
        if not (isinstance(l,AExpr) or isinstance(r,AExpr)):
            raise AExpr_Exception
        self.__rnode   = r
        self.__lnode   = l

    def left(self):
        return self.__lnode

    def right(self):
        return self.__rnode

    def __eq__(self,other):
        match other:
            case AEMult():
                return (self.__lnode == other.left() and self.__rnode == other.right())

    def __str__(self):
        return '('+str(self.__lnode)+'*'+str(self.__rnode)+')'
```

The code is quite compact and simple. It consists of a constructor `__init__` that takes arguments `l` and `r` that refer to the left and right subexpression (that must be a particular subclass of the `Expr` class). Both subexpressions can be accessed via the methods named `left` and `right`, respectively. The other methods, namely `__eq__` and `__str__` are responsible for comparing and pretty-printing the contents of the class, respectively.

The same type of hierarchy was implemented for Boolean expressions (**??**). In the case of specifications, those reflect the same structure of Boolean expressions, and the code is available in the file `Specs.py`. Similarly, the class hierarchy for the command's language is implemented in the file `Imp.py`. The class diagrams for each of these class hierarchies is presented, for your convenience, in the end of this document.

1.2 The VC_i function: reference function for completing the assignment

We will now look into the VC_i function, available in file `VCs.py`, and that implements the improved verification condition generation algorithm introduced in the classes. We present its Python code here to help your task of implementing the functions announced above.

First, we recall the algorithmic specification of the function:

$$
\begin{aligned}
VC(skip, Q) &= \emptyset \\
VC(x := e, Q) &= \emptyset \\
VC(C_1; C_2, Q) &= VC(C_1, wprec(C_2, Q)) \cup VC(C_2, Q) \\
VC(if\ B\ then\ C_1\ else\ C_2, Q) &= VC(C_1, Q) \cup VC(C_2, Q) \\
VC(while\ B\ do\ \{I\}\ C, Q) &= \{(I \wedge B) \rightarrow wprec(C, I)\} \cup VC(C, I) \\
&\quad \{(I \wedge \neg B) \rightarrow Q\}
\end{aligned}
$$

$$
VCG(\{P\}\ C\ \{Q\}) = \{P \rightarrow wprec(C, Q)\} \cup VC(C, Q)
$$

And now we look into the implemented code of the **VC_i** function:

```
def VC_i(p,pst):
    match p:
        case Skip():
            # Case: VC(skip, Q) = ∅
            return set()
        case Assgn():
            # Case: VC(x := e, Q) = ∅
            return set()
        case Seq():
            # Case: VC(C1; C2, Q) = VC(C1, wprec(C2, Q)) ∪ VC(C2, Q)
            l = VC_i(p.left(),wprec(p.right(),pst))
            r = VC_i(p.right(),pst)
            return l.union(r)
        case IfThen():
            # Case: VC(if B then C1 else C2, Q) = VC(C1, Q) ∪ VC(C2, Q)
            l = VC_i(p.left(),pst)
            r = VC_i(p.right(),pst)
            return l.union(r)
        case While():
            # Case: VC(while B do {I} C, Q) = {(I ∧ B) → wprec(C, I)} ∪ VC(C, I) ∪ {(I ∧ ¬B) → Q}
            i = { SImp(SAnd(p.inv(),bexpr2spec(p.cond())),wprec(p.body(),p.inv())) }
            j = { SImp(SAnd(p.inv(),SNeg(bexpr2spec(p.cond()))),pst) }
            r = VC_i(p.body(),p.inv())
            return i.union(j.union(r))
```

Note that the several lines starting with "`# Case:`" refers to comments that we are added here (not in the original Python file) just to make clear how the code tries to replicate what is defined at the algorithmic level of the function. This function is part of a larger function named **VCG**, thus mimicking what has been defined in the algorithmic specification. After some time looking at the produced code, it is easy to see that the function VC_i follows exactly the structure determined in the specification, that is: it first tries to understand what kind of object is is processing (via the `match` and `case` constructs) and them implements the corresponding construction of sets.

Note on sets in Python: In Python, mathematical sets are mapped into the primitive type of objects `set`. Thus the usage of the `set()` object construction to build the empty set, or the "{ }" notation to represent sets of objects. For more information on how to work with sets in Python, please refer to the following addresses:

- W3Schools: `https://www.w3schools.com/python/python_sets.asp`
- Programiz: `https://www.programiz.com/python-programming/set`
- Real Python: `https://realpython.com/python-sets/`

2 Testing your implementation

To help on the quest for the correct implementation, the code base provided includes unit test files that will allow to check if what you have implemented is indeed according to what is expected. The files are:

- `test_Wprec.py`: this file contains unit tests that verify in the weakest precondition generation function is outputting the expected results.
- `test_VCs.py`: this file contains unit tests that verify in the verification condition generation function is outputting the expected results.
- `test_Z3Driver.py`: this file contains unit tests that verify in the interface with the Z3 theorem prover is outputting the expected results.

The way to use this testing scripts is straightforward, that is, if you have access to a command line, you simply run the command `python test_Wprec.py` (and similarly with the remaining testing scripts), or you can run it directly using the "run" button available in the IDE you are using.

We now look with a bit more of detail to the structure of `test_Wprec.py` and `test_VCs.py`. The other unit testing file can be ignored, as it is not important for the objetives of the project (it is just there for those who want to install the Z3 theorem prover and see that the prover indeed is able to prove the correctness of the generated verification conditions generated by your implementation).

Interfacing with the Z3 theorem prover

As mentioned during the classes, the code that is being provided contains an interface with an external theorem prover so that we indeed can prove if programs are correct with respect to Hoare Triples. The code for this interface is available in the file `Z3Driver.py`.

To be able to use this driver, you must install the corresponding Python module. The module is named Z3Python and can be installed via the Python package manager `pip`. But you also need to install the Z3 theorem prover itself in your machine. Please follow the instructions available in the following websites:

- Windows: `https://blog.fearcat.in/a?ID=00950-982838a1-76e1-4402-9312-d3847cb98312`
- Mac (using homebrew): `https://formulae.brew.sh/formula/z3`
- Linux (using Ubuntu): `https://www.howtoinstall.me/ubuntu/18-04/z3/`

Note that if you are using a Linux distribution that is not Ubuntu, then you should use that distribution's own package manager (contact `drp@isep.ipp.pt` in case you find difficulties).

Class Diagrams for Syntax Hierarchies

The class diagram for Boolean expressions is captured in Figure 2 by the class hierarchy presented.

The class diagram presented in Fig. 3 captures the syntax of specifications. Specifications are very similar to Boolean expressions but they also consider universal and existential quantifiers. In file `Spec.py` you can find the code that translates a Boolean expression onto a logically equivalent specification; that function is named `bexp2spec`.

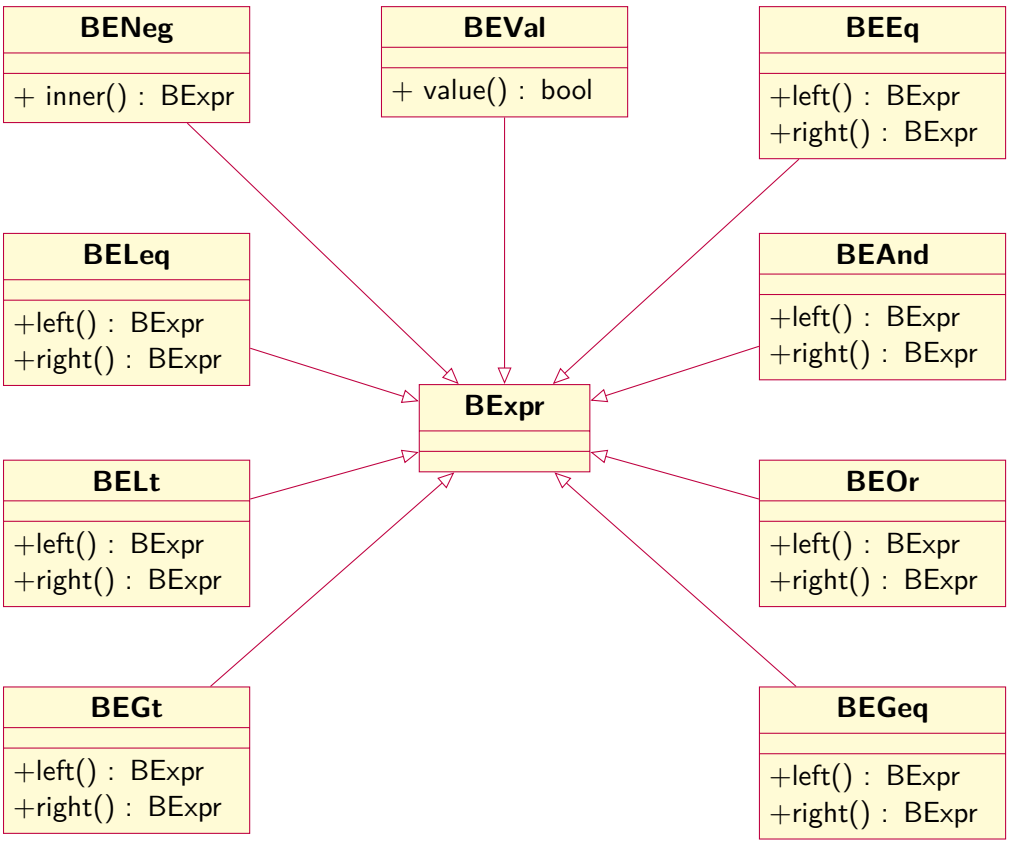

Figure 2: Boolean Expressions class hierarchy.

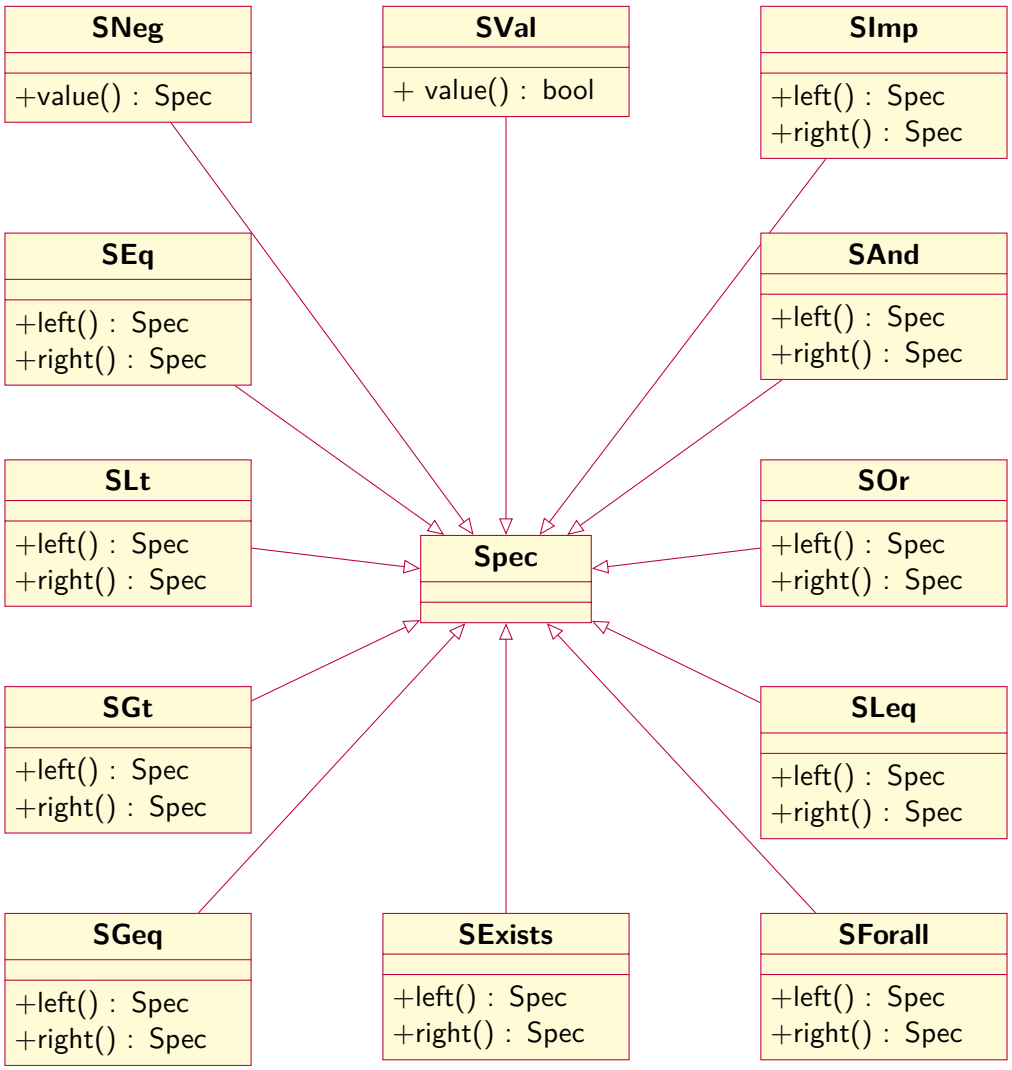

Figure 3: Specifications class hierarchy.