Assignment 2 - Verification Tool

Verification and Testing Practicals

16.11.2023

1 Introduction

For the second assignment a verification tool should be implemented which applies the Hoare logic approach presented in the lecture. As static analysis tool, it processes a given program to then prove its correctness. To achieve this, some program related aspects like `precondition`, `postcondition` as well as loop invariants and decreases clauses for proving program termination have to be supplied manually. Other statements like `assignment` or `if/else` statements are proven by the tool. In detail, within this assignment, the task is to implement the verification tool's checking procedure such that it generates and proves verification conditions for a given program. To this extend, the proving mechanism must be implemented using the `z3` solver. Please note that the required Hoare logic theory will be presented within the lecture where the termination specific aspects are mentioned in the lecture one week after this assignment's release date.

2 Disclaimer

For all assignments, use the provided files (also consider the `TODO` comments within them) and functions appropriately and do not change their names and signatures respectively. The use of additional packages is not allowed, do also not add according imports.

Furthermore, note that the project's location and current working directory should also be the path where all assignments are visible (the directory containing the `requirements.txt` file) - also run the python programs and other provided scripts from that location! When packages are stated, they are always relative to the respective assignment's package `assignmentN` where `N` corresponds to the assignment number.

Regarding the testsystem, please observe that the achieved points stated there are not final and may change. Navigating to the testsystem's results can be done by either clicking the according icon of the involved commit, or through the `Build/Jobs` overview on Gitlab.

3 Submission

For submitting your solutions, use the assigned Gitlab repository. Create a new branch `assignment-2` containing the version we should grade:

```
1  git checkout -b assignment-2
2  git push -u origin assignment-2
3  git checkout main
```

This will create a new branch callend `assignment-2` and change into that branch. After that, the new branch is pushed and you change back into the local main branch. To submit a new revision call:

```
1  git checkout assignment-2
2  git merge main
3  git push origin
4  git checkout main
```

The deadline for this assignment is `14.12.2023, 23:59`. We will grade the latest commit before the deadline on the `assignment-2` branch. You will receive a grade for the course if you submitted at least one of the assignments by creating a submission branch.

4 Framework Setup

In general, an installation of `python 3.11` with the modules described in the `requirements.txt` file is required. These modules can be installed with `pip3 install -r ./requirements.txt`.

The programming language in the framework requires `ANTLR 4`[1], a lexer and parser generation tool. Usually, since also the `antlr4_generated` directory containing the generated sources is provided, no additional steps are necessary.

However, if there are related issues, running `get_antlr4.sh` may be used for downloading `ANTLR` (consider using it with `Java 17`). Further, there is the additional script `generate_lexer_parser.sh` that can be used for generating lexer and parser for the program's grammar defined in `Program.g4`.

4.1 Docker environment

It is also possible to run the framework's assignments within a docker container. Hence, a respective `Dockerfile` is provided. Using framework and docker first requires building an image with `sudo docker build -t vt .`. The created image can be run by using `sudo docker run --rm -ti -v` `` `pwd` ``:/vt vt`. Before running these commands, make sure that the current working directory has been set to the root directory of your repository. The host system's directory containing the framework will be mounted in the container at `/vt`. Therefore, after applying `cd /vt` in the container, framework can be used as described in this document.

[1] https://www.antlr.org

5 Usage

The verification tool that has to be implemented should process programs to prove their correctness by applying the Hoare logic techniques presented in the lecture. The tool can be executed with
`python3 run_assignment2.py --file PATH`
where `PATH` corresponds to the path of the program to analyze.

5.1 Running Tests

If an individual assignment contains public tests, the command line can be used to execute them with python. More specifically, testing with the `unittest` module is achievable by typing `python3 -m unittest PATH` where `PATH` specifies the involved tests. For instance, consider that

- `python3 -m unittest assignment2.tests.public.test.Test`
  Executes all public tests contained in the assignment
- `python3 -m unittest assignment2.tests.public.test.Test.todo`
  Only runs the `todo` public test of the assignment

Note that the implementation of further tests is recommended - simply add additional functions representing test cases to the already existing ones.

6 Exercises (40 points)

The second assignment consists of multiple parts where for each one, a certain amount of points is reachable. Implement the exercise within the `verify` function of `assignment2/verification.py`. Note that the individual parts depend on each other and thus not or incorrectly implementing one part may lead to point deductions for another one. For example, a valid implementation of the `assignment` statement is also necessary to reach any points for the `if/else` and `while` statements.

6.1 Assignment statement (5 points)

The verification tool to develop should prove programs with a single function which includes verification specific aspects like a precondition and postcondition as well as different statements. Assignment statements allow programs to assign values to variables. With the provided programming language, variables can be declared by simply assigning values to them. When doing so, it is not required to state a type, since it will be derived from the assigned value.

```
1 function example1(int a, bool b)
2 precondition a == 0
3 postcondition a > 0 && !b
```

```
4 {
5     a = a + 1;
6     b = true;
7     c = 0;
8 }
```

Listing 1: A program declaring variables and assigning values to them.

For instance, listing 1 shows the function `example1` with an integer parameter `int a` and a boolean parameter `bool b`. After the function signature, the `precondition` (must hold before it is called) and `postcondition` (must hold when it terminates) are defined.

The verification should be performed using the `Z3 SMT` solver. For this assignment you have to implement the weakest precondition function for annotated programs (`pre`) as discussed in the lecture. This tasks requires the rules for assignments and sequencing. In the lecture we defined a sequencing rule for two statements you have to adapt it to work for a list of statements in a `Block` node. Using your implementation of `pre` compute the weakest precondition of the function body given the annotated postcondition. To verify if the program is correct use the SMT solver to check if the annotated precondition implies the weakest precondition computed using `pre`. The SMT tutorial on the website explains how to use Z3 to prove a formula is valid i.e. holds for all variable assignments.

In the presented example, the `postcondition` is violated and thus, the approach should find the error. More specifically, for invalid functions a
`assignment2.errors.function_verification_error.`
`FunctionVerificationError`
must be reported with the `id`, `line` and `column` from the function's node within the abstract syntax tree. Reporting errors is done by adding the error to the `errors: list[VerificationError]` parameter of the `verify` function.

6.2 If / else statement (5 points)

For control flow branching, the programming language does also implement `if/else` statements which also have to be proven. To achieve this, extend the tool to generate and check the verification conditions presented in the lecture.

```
1  function example2(int a, bool b)
2  precondition a > 0 && a < 10
3  postcondition a < 0
4  {
5      if (a > 0) {
6          a = 0;
7      } else {
8          a = 1;
9      }
10     a = a - 1;
11 }
```

Listing 2: A program using the `if/else` statement.

The usage of `if/else` statements can be seen in listing 2 where the verification tool should be able to prove the presented function to be correct. This must also work recursively: besides assignments, within the if or else branch, there may also be more complex statements such as further `if/else` statements. To implement this task add the `if/else` rule your implementation of the weakest precondition function.

6.3 Assert statement (3 points)

Assertions allow to add additional conditions to the process of verification and are used together with a boolean expression. Thus, the task is the implementation of assertion statement support for the verification tool.

```
1 function example3(int a)
2 precondition a > 0
3 postcondition a < 0
4 {
5     assert(a > 5);
6     a = -1;
7 }
```

Listing 3: A program using the `assert` statement.

A function including an `assert` statement is presented in listing 3. If implemented correct, with the assertion, the function should not be proven correct and thus a `FunctionVerificationError` must be reported. Observe that for the depicted function, removing the assertion makes the program correct. To implement this task add the `assert` rule your implementation of the weakest precondition function.

6.4 While statement

Compared to implementing verification tool support for the language statements described above, `while` loops are more complex and require dedicated effort towards proving both partial and total correctness. Develop proving mechanisms such that the tool is capable of proving the partial and total correctness of loops.

6.4.1 Partial correctness (12 points)

For partial correctness, a set of verification conditions related to an `invariant` have to be checked. Invariants are supplied manually by the programmer when writing loops. In the verification procedure it must be be proven that they strong enough and remain unchanged for all loop iterations.

```
1 function example4(int a)
2 precondition a > 5
3 postcondition a == 1
4 {
5     while (a > 0)
6     invariant a >= 0
```

```
7       {
8           a = a - 1;
9       }
10      a = a + 1;
11  }
```

Listing 4: A program using an loop `invariant` for proving partial correctness.

In listing 4, the `invariant` keyword is used for proving partial correctness for the shown function's loop. More specifically, the function is correct. Implementing this tasks consists of two parts. First, the weakest precondition function needs to be extended to handle `while` statements. Second, a loop requires addditional verification conditions that need to proven. In the lecture we discussed the function `vc` that traverses a program and produced a set of verification conditions. For every loop two verification conditions need to be generated: (1) the annotated invariant and the negated loop condition have to imply the post condition, (2) the annotated invariant and the loop condition must imply the weakest precondition of the loop body given the invariant. You can either integrate this with computing the weakest precondition or do it as a second pass over the syntax tree. In either case use Z3 to verify that both verification conditions hold.

If one of the verification conditions can not be proven the invariant is incorrect and a `assignment2.errors.invariant_verification_error.InvariantVerificationError` has to be reported. Its `line` and `column` are retrieved from the related `while` statement and its `type` parameter indicates the invariant's issue in a more detailed manner. There are two different types which are relevant:

- `assignment2.errors.invariant_verification_error.InvariantVerificationErrorType.POSTCONDITION_MISMATCH`
  When the provided invariant is not strong enough to prove the loop / program, this type must be used. For the shown function, a mismatching invariant would be `true`. This error corresponds to the first verification condition, containing the postcondition.

- `assignment2.errors.invariant_verification_error.InvariantVerificationErrorType.NO_AN_INVARIANT`
  This type indicates that the specified invariant is no invariant. As an example for the presented function, this type would be required when changing the invariant to `a >= 1`. This error corresponds to the seconf verification condition, about the loop body.

Similar to the `if/else` statement, also `while` loops have to support recursive declarations: there may also be additional loops and `if/else` statements within `while` loops.

6.4.2 Total correctness (6 points)

For total correctness, besides partial correctness, also the loop's termination has to be proven. To achieve this, the provided programming language encompasses the `decreases` keyword.

```
1  function example5(int a)
2  precondition a > 5
3  postcondition a == 0
4  {
5      while (a > 0)
6      invariant a >= 0
7      decreases a
8      {
9          a = a - 1;
10     }
11 }
```

Listing 5: A program using a loop and proving its total correctness.

For instance, in listing 5 the usage of the `invariant` keywords together with `decreases` is applied to prove the program's total correctness. Besides such simple cases, there can also be situations where the involved variable(s) for stopping the loop are not simply decreasing with each iteration.

```
1  function example6(int a, int b)
2  precondition a > 5 && b > 5
3  postcondition a == 0
4  {
5      while (a > 0)
6      invariant a >= 0 && b >= 0
7      decreases a,b
8      {
9          if (b > 0) {
10             b = b - 1;
11         } else {
12             a = a - 1;
13             b = 1;
14         }
15     }
16 }
```

Listing 6: A program using multiple `decreases` clauses within a loop

As observable in listing 6, variables may also stop decreasing. Therefore, to prove total correctness, `decreases` supports multiple clauses which together describe the decreasing behavior. Given the case where `a,b` decreases, within the verification conditions, the logical formula is $a < n_1 \vee (a = n_1 \wedge b < n_2)$. If just `a` would be decreasing, $a < n_1$ would be sufficient and the formula can be extended in that same pattern for three or more decreases clauses. Please note that thus the clause order is significant and that each clause is an expression.

To implement this task you have to generate and check the verification conditions for total correctness. The second lecture on Hoare logic defines the function `vct` to compute the verification conditions for termination. The first

condition states that $(I \wedge c) \Rightarrow E \geq 0$, and the second condition states that $(I \wedge c \wedge E = n) \Rightarrow pre(S, I \wedge E < n)$ where I is the invariant, E is the variant, c is the loop condtion, S is the loop body and n is a fresh variable.

When creating the fresh variables for the second verification condition in Z3 you have to assign them unique names. The exact naming is not important, but it is recommended to use the loop's `line`, `column` as well as the related decreases clause's index in the naming scheme, since there may be multiple `while` statements within one program.

If a loop's partial correctness was proven, but there are errors regarding total correctness, a
`assignment2.errors.termination_verification_error.TerminationVerificationError`
must be reported. Its parameters are related to the respective loop's `line`, `column` and there are two possibilities for the `type`:

- `assignment2.errors.termination_verification_error.TerminationVerificationErrorType.NOT_BOUNDED`
  This type describes that the `describes` clause(s) are not bounded. It relates to the first verification condition for termination ensuring that the variant stays positive.

- `assignment2.errors.termination_verification_error.TerminationVerificationErrorType.INVALID`
  This type describes the case where the variant is not always decreasing i.e. the second verification condition for termination is violated.

6.5 Arrays and quantifiers (4 points)

Furthermore, there is also the task to implement arrays. Within the provided programming language, arrays are already included and within the syntax tree, there are the `ListSet` and `ListGet` nodes for setting and getting an array's item at a certain index respectively. Syntax wise, writing `a[i] = i + 1` is setting the `a` array's at index `i` to `i + 1`. Semantically, this should create a new array where all items are equal to the old one except the updated one and set it to `a` - to this extend, arrays are immutable. Similarly, `b = a[i + 1]` is applied to access the array `a` at `i + 1` and assign the related value to the variable `b`.

6.5.1 Arrays (2 points)

More specifically, implementing arrays consists out of two parts. The first one is to support the `ListSet` statement and the `ListGet` statement during the verification process. For this, the `z3` solver includes a dedicated array theory which is recommended to be used for implementing the basic array functionality within the tool.

```
1 function example7(int[] a)
2 precondition a[0] == 0
3 postcondition a[0] > 0
```

```
4 {
5     a[0] = a[0] + 1;
6 }
```

Listing 7: A program using arrays.

An usage of arrays within a correct program can be seen in listing 7. In the verification process, when traversing a program, it is recommended to add encountered `ListSet` statements to the `list_sets` list of the `ListGet` expressions within the current postcondition. This is helpful when first constructing formulas by using the program's syntax tree nodes to then convert them to `z3` specific formulas. More specifically, for the `ListGet` expressions, then the collected `ListSet` statements can be used to define `z3` array formulas recursively with a combination of `Array`, `Store` and `Select`.

6.5.2 Quantifiers (2 points)

Since it is strongly related to arrays, the second task for implementing arrays within the verification tool is adding support for quantifiers. Within the programming language, within its syntax tree, there are the `Forall` and `Exists` nodes to support the respective quantifiers. Note that both are special cases of expressions and intended to be used within a function's `precondition`, `postcondition` and `invariant`.

```
1 function example8(int[] a, int l)
2 precondition l > 5 && forall i: (i >= 0 && i < l) ==> (a[i] > 0)
3 postcondition exists i: i >=0 && i < l && a[i] == 0
4 {
5     a[5] = 0;
6 }
```

Listing 8: A program using the forall and exists quantifiers.

The correct program presented in listing 5 applies the `Forall` and `Exists` quantifiers for the functions pre- and postcondition. Furthermore, it should also be implemented that multiple variables can be used in quantifier expressions. Therefore, it is required to implement that the verification tool does also support the provided programming language's capability of writing quantifiers like `forall i, j: ...`. Regarding `z3`, it is recommended to use `ForAll` and `Exists` for the definition of formulas including quantifiers.

6.6 Proofs (5 points)

Besides implementing the verification tool, another task is to prove some programs. The respective programs can be found in `assignment2.proofs`. For completing this task, please consider the instructions which are given as comments within each program. When writing the `invariant` and `decreases` clauses, replace the placeholder comments with the clauses. Do not add additional lines or remove existing ones.

6.6.1 proof_1.txt (1 point)

```
function proof3(int a)
precondition a > 5
postcondition a == 2 && i == 0
{
    i = 3;
    while (a > 0 && i > 0)
    invariant false // TODO: implement invariant
    decreases false // TODO: implement decreases clause(s)
    {
        if (a > 1) {
            a = a - 1;
        } else {
            i = i - 1;
        }
    }
    a = a + 1;
}
```

Listing 9: A program where total correctness has to be proved.

6.6.2 proof_2.txt (2 points)

```
function proof2(int[] a, int l)
precondition l > 10 && (forall i: (i >= 0 && i < l) ==> (a[i] ==
     0))
postcondition (forall i: (0 <= i && i < l && i < 5) ==> (a[i] ==
     1)) && (forall i: (0 <= i && i < l && i >= 5) ==> (a[i] ==
    2))
{
    count = 0;
    while (count < l)
    invariant false // TODO: implement invariant
    decreases false // TODO: implement decreases clause(s)
    {
        if (count < 5) {
            a[count] = 1;
        } else {
            a[count] = 2;
        }
        count = count + 1;
    }
}
```

Listing 10: A program where total correctness has to be proved.

6.6.3 proof_3.txt (2 points)

```
function proof3(int[] a, int l)
precondition forall i: (i >= 0 && i < l) ==> (a[i] > 0)
postcondition forall i: (i >= 0 && i < l) ==> (a[i] == 0)
{
```

```
5       count = 0;
6       while (count < l)
7       invariant false // TODO: implement invariant
8       decreases false // TODO: implement decreases clause(s)
9       {
10          a[count] = 0;
11          count = count + 1;
12      }
13  }
```

Listing 11: A program where total correctness has to be proved.

7 Additional information

In addition to the already described details there is some further information regarding how to properly report errors as well as some development hits and recommendations.

7.1 Error reporting

The order in which errors are reported is not significant. Consider that if there are any errors for program statements like loops a `FunctionVerificationError` must be reported. Also do not report the same errors multiple times! In addition, note that the errors indicate failing verification conditions and are not dedicated to locate errors within programs.

7.2 Implementation hints and recommendations

When processing the program and applying Hoare logic rules, it is recommended to first operate on the program's abstract syntax tree (see the classes in the `program` module) and to then convert them into formulas for consulting the solver. When doing so, it is fine to also use the syntax tree's classes to - for instance - construct expressions. The `line` and `column` parameters of new nodes can be simply set to `-1`, but make sure to report the errors correctly with `line` and `column` referencing the respective statement in the related program!

7.3 Public test cases

The assignment encompasses some public tests which should help developing a correct error reporting strategy. Further, it is recommended to add additional test cases!