Assignment 1 - Lock Tree

Verification and Testing Practicals

12.10.2023

1 Introduction

For the first and following assignments, there is a dedicated programming language developed to perform the verification exercises. Regarding the first assignment, since it focuses on dynamic analysis, there is also an interpreter for running programs. As introduction, the first assignment focuses on detecting potential deadlock scenarios by applying the lock tree algorithm.

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-1` containing the version we should grade:

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

This will create a new branch callend `assignment-1` 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:

```
git checkout assignment-1
git merge main
git push origin
git checkout main
```

The deadline for this assignment is `03.11.2023, 23:59`. We will grade the latest commit before the deadline on the `assignment-1` 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.

5 Usage

For the first assignment the lock tree algorithm presented in the lecture should be implemented within the provided framework. Appropriate errors have to be reported when executing the program. The algorithm is runnable with
`python3 run_assignment1.py --file-1 PATH-1 --file-2 PATH-2`

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

where `PATH-1` is replaced with the path to the first program's file and `PATH-2` has to be substituted with the location of the file containing the second program. This instruments and executes both programs. Each program is treated as if it where run as its own thread and one locktree will be constructed for each of them. In the end the locktrees are analized for potential deadlocks. Instrumentation, locktree construction, and dead lock detection have to be implemented as part of this assignment.

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 assignment1.lock_tree.tests.public.test.Test`
  Executes all public tests contained in the assignment
- `python3 -m unittest assignment1.lock_tree.tests.public.test.Test.test_construction_1`
  Only runs the `test_construction_1` 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 (25 points)

The assignment consists of three main parts: the lock tree construction, the actual deadlock detection and the additional task of performing instrumentation. For the involved programs, do not forget to call their functions to let the interpreter actually execute them! Please note that in order to work, the algorithm for deadlock detection depends on a correct implementation of the lock tree construction procedure.

6.1 Lock tree construction (7 points)

A requirement for actually executing the lock tree algorithm is to first construct lock trees from locking traces derived from interpreting the programs.

```
function example3() {
    lockTreeLock(l1);
    lockTreeLock(l2);
    lockTreeUnlock(l2);
    lockTreeUnlock(l1);
}
example3();
```

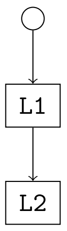
Listing 1: An example program first locking `l1` and then `l2`.

```
function example4() {
    lockTreeLock(l2);
    lockTreeLock(l1);
    lockTreeUnlock(l1);
    lockTreeUnlock(l2);
}
example4();
```

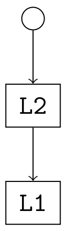
Listing 2: An example program first locking `l2` and then `l1`.

For instance, when considering the programs written within the listings 1 and 2 it is noted that calling the functions `lockTreeLock` and `lockTreeUnlock` locks and unlocks the lock passed as argument respectively.

When running the programs, these function invocations are then used to retrieve lock traces describing a program's locking behavior for a specific execution. The collected traces are then used for creating lock trees, one for each trace. Both lock trees computed from running the stated programs are shown in figure 1.



(a) Lock tree for execution of program from listing 1



(b) Lock tree for execution of program from listing 2

Figure 1: A simple lock tree example where the execution of two programs leads to a potential deadlock situation due to a lock reversal.

Further, the programs to verify may also implement an invalid locking pattern. Therefore, it is significant to recognize and report such behavior. More specifically, the following classes have to be instantiated appropriately and the resulting objects must be added to the `warnings` list:

- `lock_tree.warnings.double_locking_warning.DoubleLockingWarning`
  Double locks refer to situations where a lock that is already locked is attempted to lock another time.
- `lock_tree.warnings.invalid_locking_pattern_warning.InvalidLockingPatternWarning`
  This warning describes cases where the locking pattern itself is not valid. For instance when acquiring the locks `l1` and `l2` in that order and then

releasing `l1` the consequence is an invalid locking pattern - `l2` should have been released first.

- `lock_tree.warnings.invalid_unlocking_warning.InvalidUnlockingWarning`
  For locking traces where a lock is unlocked which is currently not locked, this warning is issued.

The lock tree itself must be constructed in the file `lock_tree.algorithm.py`. Also consider that for representing lock trees the `lock_tree.node.Node` class has to be used. For the lock tree construction, only the first warning has to be reported.

6.2 Deadlock detection (12 points)

The deadlock detection algorithm operates on valid lock trees. For its location, also choose the `algorithm.py` file. Similar to the already presented warnings which are related to invalid locking behaviors, also when - according to the lock tree algorithm - potential deadlock scenarios are found, the class
`lock_tree.warnings.deadlock_warning.DeadlockWarning`
must be instantiated correctly. Furthermore, like the other warnings, it must be added to the `warnings` list. In addition, note that for all existing potential deadlock situations appropriate warnings have to be reported.

6.3 Instrumentation (6 points)

The last exercise is the implementation of the instrumentation feature in the file `instrumentation.py`. As written in listings 1 and 2, locks are currently acquired and released by invoking the lock tree algorithm's functions. However, it may be practical to avoid these function calls and instead to rely on the lock tree algorithm being executed implicitly when operating with locks.

To achieve this, there is an additional step taking place before interpreting the program which is referred to as instrumentation. Essentially, the idea is to operate on the program's syntax tree such that the instructions to `lock` and `unlock` locks are augmented with calls to the lock tree algorithm (see listings 4 and 5). This will then derive an adapted syntax tree and the interpreter will execute the lock tree invocations to build the according lock traces.

To visualize this concept, see the program written in listing 3 where the involved locks are acquired and released directly instead of relying on function calls to the lock tree algorithm. These invocations can be imagined to take place due to performing instrumentation to still run the algorithm for construction and analysis of the related lock tree which can be observed in figure 2.

```
function example5(bool a) {
    l1.lock;
    if (a) {
      l2.lock;
      l2.unlock;
      l3.lock;
      l3.unlock;
    }
    l1.unlock;
}
example5(true);
```

Listing 3: A program which is directly acquiring and releasing the locks `l1`, `l2` and `l3`.

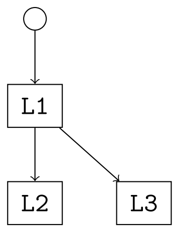


Figure 2: Lock tree constructed from the lock trace derived from executing program in listing 3.

The classes specific to program syntax tree representations can be found within the package `program` (relative to the project location). There, especially the classes `program.lock.Lock` and `program.unlock.Unlock` are relevant, since within programs, the instrumentation procedure should augment them with the `lock_tree.lock_tree_call.LockTreeCall` class. Note that there is an enum `lock_tree.lock_tree_call.LockTreeCallType` to indicate whether a lock tree call should acquire or release the respective lock.

```
function program() {
    l1.lock;
    l1.unlock;
}
```

Listing 4: A program acquiring and releasing a lock

```
function instrumented() {
    {
      l1.lock;
      lockTreeLock(l1);
    }
    {
      l1.unlock;
      lockTreeUnlock(l1);
    }
}
```

Listing 5: The program from listing 4 after instrumentation

7 Hints

For debugging, it might be useful to consider the programming language's `write` statement for generating output. As an example, `write(a);` prints the value of variable `a`.

It is recommended to use Python's pattern matching feature to implement the code instrumentation. An example for traversing a program using pattern matching can be found in the function `check_types` in `semantic_analysis.py`.