CISPA HELMHOLTZ CENTER FOR INFORMATION SECURITY

Automated Debugging

WS 2022/2023

Prof. Dr. Andreas Zeller
Paul Zhu
Marius Smytzek

Project 2 (100 Points)

Due: 21. March 2023

Submit your solutions as a Zip file on your status page in the CMS.

We will provide you with a structure to submit your solutions where each task has a dedicated file. You can add new files and scripts if you want, but you must not delete any provided ones. You can verify whether your submission is valid by calling:

```
python3 verify.py
```

The output provides an overview of whether a required file, variable, or function is missing and if you altered a function's pattern. We can only evaluate your submission if you follow this structure. A non-evaluable project will result in 0 points, so verify your work before submitting it. Note that the script only reveals if you have solutions we can evaluate.

Please use Python 3.10 to work on this project. We will use Python 3.10 for the evaluation so make sure your code runs under this version of Python.

Please read the entire project description carefully before starting to implement.

Introduction

Automated repair is a long-standing research topic of great interest in academia. The `Repairer` introduced in the chapter Repairing Code Automatically is based on GenProg [C. Le Goues et al, 2012] that uses an extended form of genetic programming to evolve a program until it passes all the test suites.

Instead of doing random mutations on the erroneous programs, is it possible to evolve (or transform) programs in more reasonable and determined ways? Answering this question brings opportunities in collaborating symbolic (and even semantic) methods into mutations. SPR [F Long et al, 2015], standing for *Staged Program Repair*, is among them. In this project, we're going to implement a proof-of-concept version of the SPR algorithm and apply it to repair buggy Python code. Although SPR involves symbolic methods, the entire framework is quite dynamic and its implementation recalls many of the key concepts and skills we've learned throughout the semester: AST instrumentation, mutations, program states tracking, invariant mining, etc.

The SPR algorithm is a repair but **not** a **defect localization** algorithm -- it requires a suspicious defect location as an input. In this project, the suspicious location (i.e., a line number) is always passed over to the repairer as an argument. In practice, this can be obtained from a statistical debugger (e.g., `OchiaiDebugger`). We won't worry about defect localization in this project.

SPR by Example

Before showing instructions on this project, let's obtain some first intuitions about the SPR algorithm using a simple example (`repair/benchmarks/char_index.py`):

In [1]:
```
from typing import Optional

def char_index(s: str, c: str) -> Optional[int]:
    """Return the index in `s` where it matches `c`.
    Special character `*` matches any character.
    Return None if not matched."""
    for i in range(len(s)):
        x = s[i]
        if x == c: # fix here
            return i

    return None
```

Note that Python does not have a `char` type (like in C, Java, etc.) so we simply annotate the type of `c` as `str` and assume it to be a string that contains a single character. When testing this function against the following test suite (`repair/benchmarks/char_index_tests.py`):

In [2]:
```
def test_1():
    """+"""
    assert char_index('ASE', 'A') == 0

def test_2():
    """+"""
    assert char_index('FSE', 'A') == None

def test_3():
    """-"""
    assert char_index('*SE', 'A') == 0

def test_4():
    """-"""
    assert char_index('F*E', 'E') == 1
```

The first two test cases (`test_1` and `test_2`) pass and the last two (`test_3` and `test_4`) fail. Following the terms used in SPR, we call the passing test cases **positive** (marked by `+` in the doc string of `test_1` and `test_2`) and the failing test cases **negative** (marked by `-` in the doc string of `test_3` and `test_4`). The big goal of program repair is to evolve the buggy program so that eventually it will pass all the test cases.

It is not hard to see the failure reason: the developer forgets that `*` is a wildcard. A possible fix is to enhance the condition `if x == c` to `if x == c or x == '*'` . This is yet another case where the `ConditionMutator` of the chapter Repairing Code Automatically is unable to fix, as the condition `x == '*'` doesn't occur anywhere in the source code.

The SPR algorithm deals with this problem in two stages -- that's why it is called **staged** program repair -- first, mutate the program using a **predetermined set of transformers** -- so pretty much similar to mutation-based repair -- but **don't concretize** any condition and instead **leave them abstract**; in the second stage, a **synthesis algorithm** guesses what the abstract condition should be.

Stage 1: Mutation

SPR defines a set of **transformation schemas** that look usual and unsurprising, including tightening or loosening an if-condition, which we've already seen in the `ConditionMutator` . Loosening the condition `if x == c` will help us repair this function:

```
def char_index(s: str, c: str) -> Optional[int]:
    for i in range(len(s)):
        x = s[i]
        if x == c or __abstract__: # <-- here mutated
            return i

    return None
```

The above is the **abstract** program that contains an **abstract** condition represented by `__abstract__` .

Stage 2: Condition Synthesis

The remaining problem is what should this `__abstract__` condition be. Well, if you are a "static guy", you may want to solve this problem using static analysis or program synthesis approaches, but these static approaches are in general not easy. Given that the test suite is supplied, SPR tackles this problem using **dynamic** approaches, namely, to run the tests on this function.

Now you may wonder how the hell SPR can execute an **abstract** program?! Obviously, an abstract program is not executable, **unless** every time this `__abstract__` condition is evaluated, some "god" tells the executor whether it evaluates to `True` or `False`. That is to say, if we provide a sequence of Boolean values, called **future abstract condition values**, to the executor, then the abstract program becomes executable (note that this `__abstract__` condition is the only thing that is abstract in the code).

For instance, let's consider the future abstract condition value sequence `[False, True]` for running the negative test case `test_4`:

1. When `i` is `0`, `x` is `'F'` so we have `x == c` evaluate to `False`. Then `__abstract__` has to be evaluated: the executor picks the first value `False`. Thus the entire if-condition evaluates to `False`.
2. In the next iteration, `i` becomes `1`, `x` is `'*'` so again `x == c` evaluates to `False`. Then, again, `__abstract__` has to be evaluated: the executor picks the second value `True`. Thus the entire if-condition evaluates to `True` and `char_index` returns `1`. Interestingly, this makes `test_4` **pass**!

This project framework includes an executor for abstract programs (the `exec_abstract` function). It records: for each evaluation of the `__abstract__` condition, (1) the evaluation result, i.e., the Boolean value consumed from the input future abstract condition value sequence, and (2) the **local environment** that records the values of all local variables (it has type `dict[str, Any]`), which is obtained by calling the Python built-in function `locals()`.

In [3]:
```
import ast
from repair.tester import exec_abstract

abstract_program = """
from typing import Optional

def char_index(s: str, c: str) -> Optional[int]:
    for i in range(len(s)):
        x = s[i]
        if x == c or __abstract__: # <-- here mutated
            return i

    return None
"""

abstract_tree = ast.parse(abstract_program)
success, record1 = exec_abstract(abstract_tree, test_4.__code__, iter([False, True]))
assert success
print(record1)
```

```
__abstract__ = False under env {'s': 'F*E', 'c': 'E', 'i': 0, 'x': 'F'}
__abstract__ = True  under env {'s': 'F*E', 'c': 'E', 'i': 1, 'x': '*'}
```

For the other negative `test_3`: the future abstract condition value sequence `[True]` is what we need.

In [4]:
```
success, record2 = exec_abstract(abstract_tree, test_3.__code__, iter([True]))
assert success
print(record2)
```

```
__abstract__ = True  under env {'s': '*SE', 'c': 'A', 'i': 0, 'x': '*'}
```

For positive test cases, the situation is trivial: all we need is the all-false sequence `[False, False, ...]`, because then the if-condition `x == c or False` will be identical to the original one `x == c`, under which the positive test cases were passed. In the following code, we simply pass over the empty sequence `iter([])` to

`exec_abstract` because we set the following: when all the elements in the sequence are consumed, the default yielding value is `False` (in other words, the empty sequence represents the all-false sequence).

In [5]:
```
success, record3 = exec_abstract(abstract_tree, test_1.__code__, iter([]))
assert success
print(record3) # this one is empty because `__abstract__` hasn't been evaluated
```

In [6]:
```
success, record4 = exec_abstract(abstract_tree, test_2.__code__, iter([]))
assert success
print(record4)
```

```
__abstract__ = False under env {'s': 'FSE', 'c': 'A', 'i': 0, 'x': 'F'}
__abstract__ = False under env {'s': 'FSE', 'c': 'A', 'i': 1, 'x': 'S'}
__abstract__ = False under env {'s': 'FSE', 'c': 'A', 'i': 2, 'x': 'E'}
```

SPR applies an **enumerate-and-check** approach to search for future abstract condition value sequences that lead to successful executions (i.e., the test case is passed): try the all-false sequence first, and if testing fails, flip the last `False` into `True` and try again. In the end, the all-true sequence is attempted. The rationale is that, in practice, if a negative test case exposes an error at an if-statement, either the last few (so we try to flip the last `False`) executions of the if-statement or all of the executions (so we try the all-true sequence) take the wrong branch direction.

If for every test case, we can find a future abstract condition value sequence that leads to successful execution, then any concrete condition `cond` that can **reproduce** the successful execution will be a candidate for instantiating the `__abstract__` condition. Back to our example, we wish to find a concrete condition that reproduces all the recorded successful executions:

In [7]:
```
print(record1 + record2 + record3 + record4)
```

```
__abstract__ = False under env {'s': 'F*E', 'c': 'E', 'i': 0, 'x': 'F'}
__abstract__ = True  under env {'s': 'F*E', 'c': 'E', 'i': 1, 'x': '*'}
__abstract__ = True  under env {'s': '*SE', 'c': 'A', 'i': 0, 'x': '*'}
__abstract__ = False under env {'s': 'FSE', 'c': 'A', 'i': 0, 'x': 'F'}
__abstract__ = False under env {'s': 'FSE', 'c': 'A', 'i': 1, 'x': 'S'}
__abstract__ = False under env {'s': 'FSE', 'c': 'A', 'i': 2, 'x': 'E'}
```

Such a concrete condition is called a **satisfiable solution**. Formally, we say `cond` is a satisfiable solution of a given `record`, if for every item `value, env`, `cond` evaluates exactly to `value` under the environment `env`.

To synthesize a satisfiable solution, SPR again applies an enumerative-and-check approach: it first constructs a set of **candidate conditions** and returns any that is satisfiable (i.e., reproduces all the successful executions). By default, SPR considers candidate conditions of form `x == v` and `x != v` for all items `x : v` (`x` is a variable and `v` is its value) seen in the recorded environments. For our example, these candidate conditions are:

```
s == 'F*E'  s == '*SE'  s == 'FSE'
c == 'E'    c == 'A'
i == 0      i == 1      i == 2
x == 'F'    x == '*'    x == 'S'    x == 'E'
```

and their `!=`-variants.

It is possible to also consider other candidates instantiated from predefined templates, just like the invariant templates we've seen in the chapter Mining Function Specifications. But for our example, a satisfiable solution -- `x == '*'` -- is already among the default candidate conditions; so we don't have to consider other templates.

Finally, instantiating `__abstract__` with `x == '*'`, we obtain a new version of `char_index`:

In [8]:
```
def char_index(s: str, c: str) -> Optional[int]:
    """Return the index in `s` where it matches `c`.
    Special character `*` matches any character.
    Return None if not matched."""
```

```
    for i in range(len(s)):
        x = s[i]
        if x == c or x == '*': # <-- instantiated
            return i

    return None
```

which can pass all the supplied tests:

In [9]:
```
test_1()
test_2()
test_3()
test_4()
```

Grading

In this project, you are not going to implement SPR from scratch by yourself, but **fill in the missing methods and classes** (search for `# TODO: YOUR CODE HERE`) in the provided source folder **`repair/`**. The basic framework of the SPR algorithm is given.

When editing code, remember:

1. Don't change the folder structure.
2. Don't delete any existing file/class/method/variable.
3. Don't change the type signature of any method/variable.
4. You are allowed to add helper classes/methods/variables and new files at your convenience.
5. You are allowed (and recommended) to add your own tests in `repair/tests/` and benchmark programs in `repair/benchmarks/`. Note that our grader will not test on your test cases but only ours.
6. If necessary, you can modify code in `repair/mutator.py` and `repair/synthesizer.py` (there is nothing you need to implement in the other files so don't change them) that are not marked as `# TODO: YOUR CODE HERE` as long as you understand them and have confidence you don't break things already built there.

The public tests are provided in `repair/tests/`. Launch Python's `unittest` library to run them (as you did in project 1):

```
python3.10 -m unittest <path>
```

If you pass all the public tests, you will get **82 out of 100** points. The other 18 points come from hidden tests -- the hidden tests only apply to Task 3; all the tests that apply to Task 1 and Task 2 are public.

Each tests has a **time limit** (timeout gives you 0 points). For tests in `repair/tests/test_repairer.py` (they are end-to-end tests), each has a time limit of 30 seconds. The others are unit tests and each has a time limit of 10 seconds.

Task 1: Mutation (40 Points)

For this part, all the classes and methods you need to implement are in file `repair/mutator.py`. Run the tests in file `repair/tests/test_mutator.py` to check your implementation -- each test is worth **1 point**.

Marker

The first stage of SPR is program mutation. To implement this smoothly, let's first implement a helper class `Marker` that seeks a statement `node` (of type `ast.stmt`) whose line number equals the given defect location `line_no`. If found, this `node` is called the **target** -- where we are going to mutate. By marking, we mean to set an attribute named `__target__` on this `node`, and its values is a tuple `(in_loop_body, is_first_stmt)` consisting of:

1. a Boolean `in_loop_body` indicates whether this `node` is inside a loop body;
2. a Boolean `is_first_stmt` indicates whether this `node` is the first statement in its block (direct parent).

In [10]:
```
import ast
```

```
class Marker(ast.NodeTransformer):
    """Mark the target statement."""
    def __init__(self, line_no: int) -> None:
        super().__init__()

        self.line_no = line_no
        self.found = False          # target found?
        self.loop_level = 0         # depth of loop (0 indicates outside loop body)
        self.is_first_stmt = False  # is the first stmt in block?

    def generic_visit(self, node: ast.AST) -> ast.AST:
        if isinstance(node, ast.stmt) and node.lineno == self.line_no:
            setattr(node, '__target__', (self.loop_level > 0, self.is_first_stmt))
            self.found = True
            return node

        # TODO: YOUR CODE HERE
```

The `generic_visit` function already realizes the `__target__` attribute setting. Your task is to modify the rest code (copied from the super class implementation) to correctly maintain two state variables:

1. `self.loop_level`: How many levels of loops we have entered so far (increase it when a loop body is entered and decrease it when exited).
2. `self.is_first_stmt`: Whether the currently visited node (if it is a statement) is the first statement of its code block.

Once you've done this step, run the tests in classes `repair.tests.test_mutator.TestMarker*`.

Mutation Operators

SPR involves plenty of transformation schemas. In this project, we consider the following four schemas that introduce abstract conditions:

- **Tighten**: If the target statement is an if-statement, transform its condition by conjoining an (inverted) abstract condition, i.e., transform `if c` into `if c and not __abstract__`.
- **Loosen**: If the target statement is an if-statement, transform its condition by disjoining an abstract condition, i.e., transform `if c` into `if c or __abstract__`.
- **Guard**: Wrap the target statement with an (inverted) abstract condition, i.e., transform `s` into s => `if not __abstract__: s`.
- **Break**: If the target statement is in a loop body, insert a `break` statement right before it that executes only if an abstract condition is true, i.e., `if __abstract__: break`.

Note that some of the schemas introduce negations (`not __abstract__`): this is **intentional** to make sure when `__abstract__` takes `False`, the transformed program is **semantically identical** to the original.

Each of the above is defined as a subclass of `MutationOperator` -- itself is a subclass of `ast.NodeTransformer`:

In [11]:
```
class MutationOperator(ast.NodeTransformer):
    def __init__(self) -> None:
        super().__init__()
        self.mutated = False
```

Implement the four classes (`Tighten`, `Loosen`, `Guard`, and `Break`) according to the description above. These classes will be called after the tree is marked (i.e., by applying the `Marker`). Thus, you should assume the tree contains a node (of type `ast.stmt`) with the `__target__` attribute, and you are free to extract the Boolean properties `in_loop_body` and `is_first_stmt` from its value. If the mutation is happened, set `self.mutated` to `True`.

In the `Break` class, the constructor parameter `required_position` specifies an additional constraint on when this operation should be applied:

- If the value of `required_position` is `True`, this operation is applied only when the target statement is the first statement (i.e., `node.__target__[1] == True`).
- Else, this operation is applied only when the target statement is not the first statement (i.e., `node.__target__[1] == False`).

We make the distinction here because these two cases have **different priorities** in SPR. The priority orders for the transformation schemas considered here are: `Tighten() > Loosen() > Break(True) > Guard() > Break(False)`. That is,

- the algorithm tries to tighten first;
- If fails, it tries to loosen;
- If fails, it tries to insert a conditional break-statement if the target is the first statement;
- If fails, it tries to insert a guard;
- If fails, it tries to insert a conditional break-statement if the target is not the first statement;
- If all of the above fails, the entire repairing algorithm fails.

The `Mutator` class applies transformation schemas in the above order by default. Each yielded tree contains an abstract condition and will be sent to the condition synthesis algorithm for solving the condition. Suppose the condition synthesis fails, SPR will proceed to try the next transformation schema.

Once you've done this step, run the remaining tests in `repair.tests.test_mutator`. Each operator named `<Op>` has its own test class `Test<Op>`, so you can test your operator classes one by one.

Task 2: Condition Synthesis (30 Points)

For this part, you need to implement a few methods defined in the `Synthesizer` class in file `repair/synthesizer.py`. Run the tests in file `repair/tests/test_synthesizer.py` to check your implementation -- each test is worth **1 point**.

The main algorithm is already implemented in the `apply` method. First, read the code to better understand the process of enumerating future abstract condition values. The argument `k` restricts the maximum rounds of attempts for each negative test case. Next, follow the instructions below to implement the `flip` and `solve` methods.

Flip

The flip operation returns a new list where the last `False` in the old list is flipped to `True`, and all the `True`s in the old list afterward are dropped. See `repair.tests.test_synthesizer.TestFlip` for concrete examples.

Formally, this flip operation can be defined as a recursive function:

```
flip([]) = []
flip(r + [False]) = r + [True]
flip(r + [True]) = flip(r)
```

Implement the `flip` method in the `Synthesizer` class. Once you've done this step, run the tests in `repair.tests.test_synthesizer.TestFlip`.

Optionally, you can enable the `log` option for this class and run the tests in `repair.tests.test_synthesizer.TestSynthesizerCharIndex` to see which value sequences are tried for each test case (of course these tests still fail because the `solve` method is yet not implemented). In the `apply` method, the `overall_record` maintains the meta-data of successful executions: the values of the `__abstract__` condition that were taken, together with the local environments. If you are curious, see the code in `tester.py` (but to finish this project only, you don't need to look at this file).

Solve

The actual condition synthesis is performed in the `solve` method `def solve(self, constraints: Record) -> bool`. Your goal is to find a satisfiable solution `cond: ast.expr` of `constraints`, meaning it can reproduce the successful executions recorded in `constraints`. If succeed, set `self.condition` to the satisfiable solution and return `True`; otherwise, return `False`. You are asked to find **only one** satisfiable condition (if exists). There is a helper method `def sat(self, cond: ast.expr, constraints: Record) -> bool` that checks if a `cond` is satisfiable. Just call it each time you find a candidate.

The candidate enumeration consists of two phases. In the first phase, only enumerate candidate conditions of form `x == v` and `x != v` for all items `x : v` seen in the environments of `constraints`. Once you've done this step, run the tests in `repair.tests.test_synthesizer.TestSolveBasic`.

If no satisfiable condition is found, then go to the second phase: enumerate candidate conditions from a set of user-specified condition templates (when `self.extra_templates` is not `None`).

In [12]:
```
from typing import List, Tuple

class Template:
    """Condition template."""
    def __init__(self, meta_vars: List[Tuple[str, str]], body: ast.expr) -> None:
        self.args: List[str] = [s for s, _ in meta_vars]
        self.types: List[str] = [t for _, t in meta_vars]
        self.body = body

    @classmethod
    def from_lambda(cls, expr: str):
        """Parse a template from a string, basic syntax:
        (<var> : <type>, <var> : <type>, ...) => <condition>
        The lhs parenthesis is optional."""
        ...

    def instantiate(self, vars: List[str]) -> ast.expr:
        """Instantiate a template with `vars`. Just like applying values to
        lambda expressions."""
        ...
```

Templates are encoded as lambda expressions. For example, the condition template `X > Y` (using the syntax introduced in Mining Function Specifications) is encoded as the lambda expression `(X: int, Y: int) => X > Y`, where `X` and `Y` are arguments, and they both have type `int`.

In [13]:
```
from repair.synthesizer import Template

# create a template from lambda expression syntax
# NOTE: This is not the Python's lambda syntax but rather
# Scala's syntax + Python expression in the body.
# Python's lambda does not allow type annotations on args.
template = Template.from_lambda('(X: int, Y: int) => X > Y')

print(f'args = {template.args}')
print(f'types = {template.types}')
```

```
args = ['X', 'Y']
types = ['int', 'int']
```

The types are represented by their names obtained via calling `<type>.__name__`, e.g.:

In [14]:
```
int.__name__
```

Out[14]:
```
'int'
```

In [15]:
```
type([1, 2, 3]).__name__
```

Out[15]:
```
'list'
```

The body `X > Y` is encoded as an AST (of type `ast.expr`):

In [16]:
```
print(ast.dump(template.body, indent=2))
```

```
Compare(
  left=Name(id='X', ctx=Load()),
  ops=[
    Gt()],
  comparators=[
    Name(id='Y', ctx=Load())])
```

To obtain a concrete condition (of type `ast.expr`), you must instantiate a template with actual variable names, such as instantiating `template` with `a` and `b` (assuming they both have type `int`):

In [17]:
```
e = template.instantiate(['a', 'b'])
print(ast.unparse(e))
```

```
a > b
```

For each condition template in `self.extra_templates`, enumerate all possible instantiations with variables seen in the environments of `constraints`. To save the potentially large number of attempts, only consider instantiations that are **type-compatible**. For example, instantiate `(X: int, Y: int) => X > Y` with variables that are integers but not strings, lists, etc.

Once you've done this step, run the tests in `repair.tests.test_synthesizer.TestSolveTemplates`. Also, run the remaining tests in `repair.tests.test_synthesizer` to check the entire synthesis algorithm.

Task 3: Putting it All Together (30 Points)

For this part, you don't need to implement anything, but just run the tests in file `repair/tests/test_repairer.py` to check if the overall repairing algorithm (Mutator + Synthesizer) works.

There are in total 10 test cases to examine the correctness of your repairer. The file `repair/tests/test_repairer.py` includes 4 out of the 10; the other 6 are hidden. We recommend you come up with your own tests that cover the missing features of public tests, such as other transformation schemas and condition synthesis using user-specified templates.

Each test case is worth **3 points**:

- You get 3 points if all the tests (positive + negative) are passed.
- You get 2 points if all the positive tests are passed and at least one negative test is passed.
- You get 1 point if all the positive tests are passed but all the negative tests are failed.
- Otherwise, you get 0 points (in other words, you break at least one positive test).

Note that the test cases will be run **only when** your repairer **executes normally**; otherwise, you get 0 points. The following are some typical situations where your repairer runs abnormally:

- It raises any uncaught exceptions;
- It times out (time limit 30 seconds);
- The repaired AST is not generated (say, it is `None` or even not defined).