Application of dynamic slicing analysis in Python

Li Lin, st185873

st185873@stud.uni-stuttgart.de

February 1, 2024

This report introduces a project to implement dynamic backward slicing analysis in Python based on the DynaPyt[1] framework. This report outlines the background, objectives, implementation methods, shortcomings, and final results of the project. The project code repository is at https://github.com/linli-cl/Dynamic_Slicing_DynaPyt.git.

1 Introduction

1.1 Project Goals

The project aims to extract a subset of code from a given Python program that is relevant to specific slicing criteria through dynamic analysis techniques. The scope of the analysis is within the execution of a Python function.
The analysis of this project is mainly divided into two milestones, data flow slicing and data flow plus control flow slicing. Generally speaking, data flow is statements with a sequential structure in Python, and control flow is statements with a conditional or loop structure.

1.2 Dynamic analysis slicing algorithm

Program slicing is a program analysis technique that aids in debugging and understanding programs by extracting subsets of code that are relevant to specific slicing criteria.

Forward vs. backward Forward slicing is a slicing of statements influenced by the slicing criteria. Backward slicing is slicing of statements that influence the slicing criteria.

Static vs. dynamic Static slicing is performed on the static structure of the program, regardless of the actual execution of the program. Dynamic slicing is performed during program execution, slicing the code that has been used relative to the slicing criteria based on execution history.

Algorithm used: graph reachability problem of PDG(Program Dependence Graph) The nodes of the graph represent statements, and the edges of graph represent data flow and control flow dependencies. Data dependencies are defined to find Definition-Use pairs between different statements, where defined variable affects used variable. Control dependencies have a more complex definition, which is simplified in this project to the dependencies between the control statements of if, for, and while and the corresponding controlled statements. Based on PDG, slice all statements that influence the slicing criteria, that is, the graph reachability problem of finding all nodes that can reach the node of the slicing criteria.

1.3 Tools and Libraries

DynaPyt This library provides a set of hooks for capturing different runtime events during program execution, such as variable reading and writing, Boolean expressions, and conditional judgments[2]. The capture of these events provides the basis for dynamic slicing analysis. When using DynaPyt, the code needs to be instrumented first and then analyzed.

LibCST[3] This library provides operations on Python AST(Abstract Syntax Tree), making code parsing, manipulating and reprinting easier.

2 Dynamic analysis implementation

To implement the slicing algorithm mentioned above, the main problem to be solved is: how to create PDG? Or, how to find the nodes and edges of PDG? This can be achieved through python dictionaries. The implementation of this project can be summarized into three steps:

1. Find the nodes: Capture the runtime event through the hooks of DynaPyt, and use Python dictionaries to store the code statements line numbers and variable names.
2. Find the edges: Obtain the line numbers of the statements to be sliced by performing a recursive search on the Python dictionaries.
3. Modified source code: Based on the retained line numbers, manipulate AST through LibCST and return the modified code.

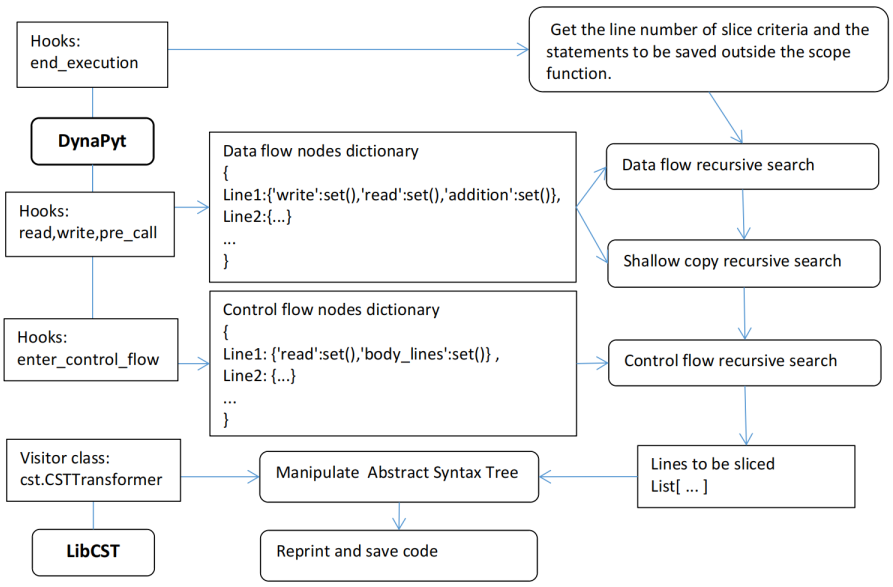

Figure 1: Slicing progress.

2.1 Events hooks and nodes storage

In the class that implements slicing, inherit DynaPyt's BaseAnalysis and rewrite each hook mentioned below. The specific rewriting is to analyze the AST of the event node to find its line number and variable names and write it into the Python dictionary after the hook captures the corresponding event.

Events hooks The following are the hooks that were used.

Basic: *end_execution*;

For data flow: *read, write, pre_call*;

For control flow:*enter_control_flow*.

Nodes stroage The Python dictionary structure is as follows. Each key-value pair:
For data flow: *line_number: 'write':set(),'read':set(),'addition':set()* ;
For control flow: *line_number: 'read':set(),'body_lines':set()* .
"read" stores the variable name that is read. "write" stores the variable name that is written. "addition" stores the variable name of the shallow copy of the class object or list. "body_lines" stores the line numbers of all the controlled statements of the corresponding control statement.

2.2 Recursive search

We now have the variable names and location information for the event nodes, as well as the slice criteria. We need to find all nodes' definition-use pairs related to it.

For data flow slice The recursive search is implemented in the rewrite of the end_execution hook. Starting from the defined(written) variable name of slice criteria and searching for its used(read) variable name. Based on this used variable name, reverse recursion to find its defined variable and required used variables. Recurse until the first defined variable. Save the line numbers of all recursive statements to obtain the slice results of the data flow.

For shallow copy situation Search "addition" sequentially. If the variable name in the addition is a variable name that exists in the data flow slice result, the same recursive search will be performed starting from the defined variable name of this statement. Add the result to the slice result.

For control flow slice Search the control flow nodes storage dictionary in reverse order. Determine whether the body lines of the control statement exist in the above slicing results. If so, the same recursive search will be performed starting from this statement's used variable.

```
def slicepoint(self,statements_line,slice_point):
    '''
    After getting the graph. Use recursion to get slice nodes.
    '''
    self.slice_results_line.add(statements_line[0]) # here got the slice line
    for j in slice_point: # continue the recrusion of slicing
        for k in range(1,len(statements_line)):
            if j in self.graph_nodes[statements_line[k]]['write']:
                self.slicepoint(statements_line[k:],self.graph_nodes[statements_line[k]]['read'])
```

Figure 2: Code to implement recursive search.

After the recursive search is completed, the line numbers of all statements that need to be sliced are obtained.

2.3 Manipulate AST and reprint code

Generate AST through LibCST parsing code. Inherit the CSTTransformer basic visitor class, traverse the nodes of the tree, and determine whether the node is "SimpleStatementLine", "nodes.statement.If", "nodes.statement.While" or "nodes.statement.For", and whether the node line number is not in the slice result. If so, remove this subtree.
For the "Else" statement, additional judgment is required because it is not stored in the node dictionary of the control flow. Determine whether the node is "nodes.statement.Else" and its body lines after its line number are in the slice result. If not, remove the subtree.
Finally, the modified tree is reprinted as code and saved in a file.

3 Shortcomings

The project scope is limited to one function. It has not yet been implemented to identify classes or functions outside of scope function.

```
1   class Person:
2       def __init__(self, name):
3           self.name = name
4           self.age = 0
5
6       def increase_age(self, years):
7           self.age += years
8
9   def slice_me():
10      p = Person('Nobody')
11      while p.age < 18:
12          p.increase_age(1)
13      if p.age == 18:
14          print(f'{p.name} is {p.age}')
15      return p # slicing criterion
16
17  slice_me()
```

Figure 3: Test case in milestone3 test_7

In the example of the figure 3, the code on line 12 calls the code on line 7. The code in line 7 implements the writing of the variable ”age”, but the write hook does not capture the write event in line 12. Need to be captured through additional hooks. In this project, pre_call hook is used to capture and identify whether the function name is ”Person.increase_age”. But this greatly reduces the generalization ability of code slicing. If the class name changes, it is not feasible.

4 Testing results

```
(prog_analysis) root@LinLi:/home/liln/23w_program_analysis/myproject/Dynamic_Slicing_DynaPyt# pytest tests --only tests/milestone2
============================= test session starts ==============================
platform linux -- Python 3.10.0, pytest-7.4.3, pluggy-1.3.0
rootdir: /home/liln/23w_program_analysis/myproject/Dynamic_Slicing_DynaPyt
collected 13 items

tests/run_single_test.py .............                                   [100%]

============================== 13 passed in 1.32s ==============================
(prog_analysis) root@LinLi:/home/liln/23w_program_analysis/myproject/Dynamic_Slicing_DynaPyt# pytest tests --only tests/milestone3
============================= test session starts ==============================
platform linux -- Python 3.10.0, pytest-7.4.3, pluggy-1.3.0
rootdir: /home/liln/23w_program_analysis/myproject/Dynamic_Slicing_DynaPyt
collected 11 items

tests/run_single_test.py ...........                                    [100%]

============================== 11 passed in 13.98s ==============================
```

Figure 4: Test results

All test examples were successfully sliced.

References

[1] Aryaz Eghbali and Michael Pradel. *https://github.com/sola-st/DynaPyt*, 2022.

[2] Aryaz Eghbali and Michael Pradel. Dynapyt: A dynamic analysis framework for python. *ESEC/FSE 2022: Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering,09 November 2022*.

[3] *https://github.com/Instagram/LibCST*.