Dynamic Python program analysis using backwards slicing

FELIX NEUBAUER, University of Stuttgart, Germany

This report deals with the implementation of backwards slicing based on the dynamic execution of Python programs.

1 INTRODUCTION

Slicing is a technique used in the field of program analysis to extract a sub-program from a given source code by neglecting parts of this code which have no influence on the behavior that is to be analysed. If, for example, we want to analyze why a particular integer variable in the code is assigned to the value that it is, we could use slicing to extract a sub-program, which neglects every other part of the code which has no influence on the value of this particular integer variable. Slicing can be static (without running the code) or dynamic (based on an actual execution of the code). Static slicing tends to over-approximate the slice because it includes every possible dependency, while dynamic slicing tends to under-approximate the slice, since it only considers the code that was part of a particular execution. Additionally, we differentiate between backward slicing (concerned with the code that result in a certain state) and forward slicing (predict parts of the programs that will be affected by a state). This report is about the implementation of dynamic backwards slicing for Python programs.

2 TASK

The task is defined as follows:

> "The goal of this project is to implement **dynamic backward slicing** for Python. [...] Given a Python program and a slicing criterion, the final output of your analysis should keep only the code needed for the slice, e.g., by removing subtrees from the AST that are not part of the slice. [...]" [1]

Listing 1 shows an example input Python code and listing 2 shows how the slice should look like that would be extracted from this piece of input code.

Code Listing 1. Example Input Code

```
def slice_me():
    a = 1
    b = 2
    a = a + 3 # slicing criterion
    a += 2
    return a

slice_me()
```

Code Listing 2. Corresponding Output Slice

```
def slice_me():
    a = 1
    a = a + 3 # slicing criterion

slice_me()
```

3 APPROACH

Using the Python library DynaPyt[2], the given source code is instructed with hooks that the program slicing code has access to. We deploy hooks for the Python code execution events `write` , `read` , `pre_call` . All of those events are recorded during dynamic program execution, storing the type of event, associated variables and the line of the corresponding code. After the program execution, a data flow analysis is run on the data structure of recorded events to

[1] *Task Description:* https://software-lab.org/teaching/winter2023/pa/project.pdf
[2] *DynaPyt: A Dynamic Analysis Framework for Python:* https://github.com/sola-st/DynaPyt

Author's address: Felix Neubauer, felix@neuby.de, University of Stuttgart, Universitätsstr. 38, Stuttgart, BW, Germany, 70569.

construct a dependency graph based on the data flow. Static code analysis techniques are used to determine structural dependencies (e.g. every line within the body of a class or function depends on the head of the class or function definition), which are also integrated into the dependency graph. Control flow dependencies are represented in a similar manner: for every control flow element, a `depends_on` relationship is added for each child of the control flow element to the control flow element (e.g. every line in the body of an if condition depends on the if condition line). This way, a line of code that deals with control flow is kept in the slice if either A) the line is relevant because of data flow dependencies or B) a child line is relevant because of data flow dependencies.

3.1 Event recording

Using the DynaPyt hooks `write` , `read` , `pre_call`, the following kinds of events are detected and recorded:

- EventAssign: when a variable is being assigned to (excludes augmented assignment), resulting in a previous value (if existing) being overwritten.
- EventUse: when a variable is being read.
- EventModify: when a object is being modified, for example by augmented assignment or because a modification/assignment is performed on a child object.
- EventAlias: when an object is assigned to another variable.

Listings 3 and 4 demonstrate which code triggers the different events.

Code Listing 3. Example for recorded events

```
a = 5
# EventAssign(var="a", line=1)

a += 2
# EventModify(var="a", line=4)

b = a
# EventUse(var="a", line=7)
# EventAssign(var="b", line=7)
# EventAlias(alias="b", var="a", line=7) <- suboptimal

example_dict = {"x": 1, "y": 2}
# EventAssign(var="example_dict", line=12)

example_dict["z"] = 3
# EventAssign(var="example_dict[?]", line=15)
# EventModify(var="example_dict", line=15)

example_dict["x"] += 10
# EventUse(var="example_dict", line=19)
# EventModify(var="example_dict[?]", line=19)
# EventModify(var="example_dict", line=19)

class Person:
    def __init__(self, age):
        self.age = age
        # EventUse(var="age", line=26)
        # EventAssign(var="self.age", line=26)
        # EventModify(var="self", line=26)
        # EventAlias(alias="self.age", var="age", line=26)
```

Code Listing 4. Example for recorded events continuation

```
p = Person(10)
# EventUse(var="Person", line=32)
# EventAssign(var="p", line=32)

p.age = 11
# EventAssign(var="p.age", line=36)
# EventModify(var="p", line=36)

p.age += 5
# EventUse(var="p", line=40)
# EventUse(var="p.age", line=40)
# EventModify(var="p.age", line=40)
# EventModify(var="p", line=40)

p2 = p
# EventUse(var="p", line=46)
# EventAssign(var="p2", line=46)
# EventAlias(alias="p2", var="p", line=46)

p2.age = 4
# EventAssign(var="p2.age", line=51)
# EventModify(var="p2", line=51)
```

3.2 Dataflow analysis

After the instructed code was executed, the recorded events are used to generate a dataflow dependency graph. This is done by iterating over all the events `e` and performing the following logic:

(a) EventAssign: in a data structure store that the lasted assignment for `e.var` was in line `e.line`. If `e.var` was an alias for another variable: delete this alias connection.
(b) EventUse: retrieve a list of locations where `e.var` was previously defined (from the data structure of latest assignments, or from a list of class and function definitions, which has been extracted using static code analysis), also considering aliases. For every of these definition locations, add a definition-use edge to the graph using `definition.line` and `e.line` as nodes.
(c) EventModify: Identical to the logic from EventUse and EventAssign combined, except it does not delete alias connections.
(d) EventAlias: Registers the alias connection in a data structure.

This way, the dataflow graph is populated.

3.3 Controlflow and structural analysis

Controlflow analysis could be performed in a similar manner as dataflow analysis, if we would also record controlflow events. However, we can also make use of the dataflow graph that is already computed and extend it. Controlflow elements (`if`, `else`, `while`, `try`, `with`, ...) are relevant for the slice only if either a) they themselves are a dataflow dependency (e.g., by modifying an object within the condition code) or b) a child element in the AST is a relevant controlflow or dataflow dependency. Therefore, to model controlflow dependencies, it is enough to add a `has-dependent` edge from every child of a controlflow element to the controlflow element itself. This models even complex scenarios, as shown in section 4.

The same technique is applied on class definitions and function definitions: a `has-dependent` edge is added from each line of the class/function body to the line of the class/function header. This way, as soon at least one line within a function or class is relevant for dataflow, the function/class header is included in the slice too. We also add one edge from each method header to the class definition header, to ensure that all functions of classes are included in the slice.

3.4 Slice computation

The graph that is built up in sections 3.2, 3.3 is an RDF[3] knowledge graph. After it is fully populated with dataflow, controlflow and structural dependencies, a SPARQL[4] query is used to receive all nodes (lines of the source code) which are connected with the *slicing criterion* node directly or indirectly with incoming edges. The slicing criterion line is determined using the libcst[5] Python library.

4 EVALUATION

In the scenarios shown in appendix A the implemented backwards slicing technique behaves the way it is supposed to. The implemented approach has the limitations listed in the task description[6] (over-approximation of dataflow for modifications and read access on complex objects, only intra-procedural analysis, etc.).

[3] *Resource Definition Framework (RDF):* https://www.w3.org/RDF/
[4] *SPARQL Query Language:* https://www.w3.org/TR/sparql11-query/
[5] *libcst:* https://libcst.readthedocs.io
[6] *Task Description:* https://software-lab.org/teaching/winter2023/pa/project.pdf

Additionally, it over-approximates aliases. Whenever the value of an existing variable is assigned to another variable in program execution, this approach considers that as the creation of an alias. This makes sense when the object behind the variable is a class instance or container (e.g., dict, list), but not for simple types (e.g., int, string). Listing 5 shows an example source code and listing 6 the resulting (suboptimal) slice, while listing 7 shows the optimal minimal slice. This could be avoided by implementing a technique to distinguish between simple and complex types in assignments.

Code Listing 5. Source code

```
def slice_me():
    a = 5
    b = a
    b += 2
    return a # slicing criterion
slice_me()
```

Code Listing 6. Slice

```
def slice_me():
    a = 5
    b = a
    b += 2
    return a # slicing criterion
slice_me()
```

Code Listing 7. Optimal slice

```
def slice_me():
    a = 5
    return a # slicing criterion
slice_me()
```

5 CONCLUSION

Dynamic backwards slicing for Python programs is successfully implemented within the given scope. Dataflow dependencies are resolved by executing the instructed program and recording dataflow events, on which then dataflow analysis is performed. Controlflow dependencies are resolved using the results from dataflow analysis and a certain logic, bypassing the need for recording controlflow events. The evaluation shows that the implemented slicing technique behaves the way it is supposed to, with the exception of the over-approximation of aliases.

A SCENARIO EVALUATION

A.1 Scenario *Dataflow with simple types*

Code Listing 8. Source code

```
1  def slice_me():
2      a = 1
3      b = 2
4      c = 3
5      d = a + b
6      e = a + c
7      x = 0
8      y = 0
9      x += d
10     y += d
11     return x # slicing criterion
12 slice_me()
```

Code Listing 9. Slice

```
def slice_me():
    a = 1
    b = 2
    d = a + b
    x = 0
    x += d
    return x # slicing criterion
slice_me()
```

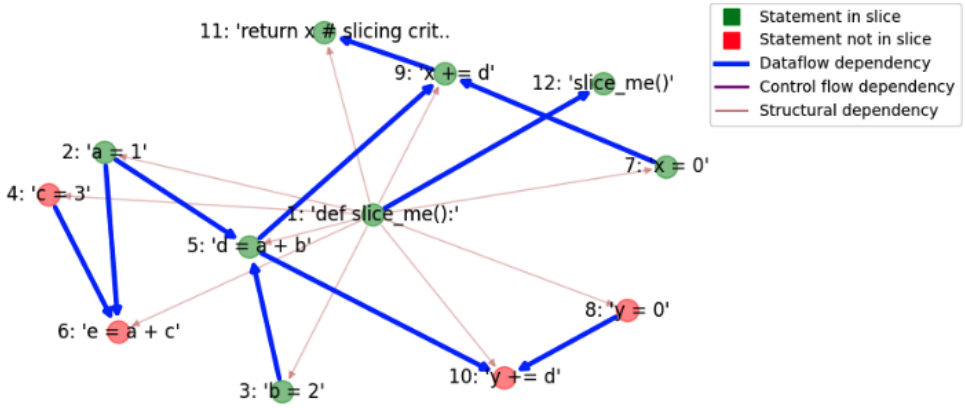

Fig. 1. Dependency graph

A.2 Scenario *Dataflow with classes and containers*

Code Listing 10. Source code

```
1  class Person:
2      def __init__(self, age):
3          self.age = age
4  def slice_me():
5      p1 = Person(1)
6      p2 = p1
7      p2.age += 5
8      text = "p1 age is " + str(p1.age)
9      unused_list = [1, 2, 3]
10     unused_list.append(text)
11     result = ["text1"]
12     result.append("text2")
13     result.append(text)
14     return result # slicing criterion
15 slice_me()
```

Code Listing 11. Slice

```
class Person:
    def __init__(self, age):
        self.age = age
def slice_me():
    p1 = Person(1)
    p2 = p1
    p2.age += 5
    text = "p1 age is " + str(p1.age)
    result = ["text1"]
    result.append("text2")
    result.append(text)
    return result # slicing criterion
slice_me()
```

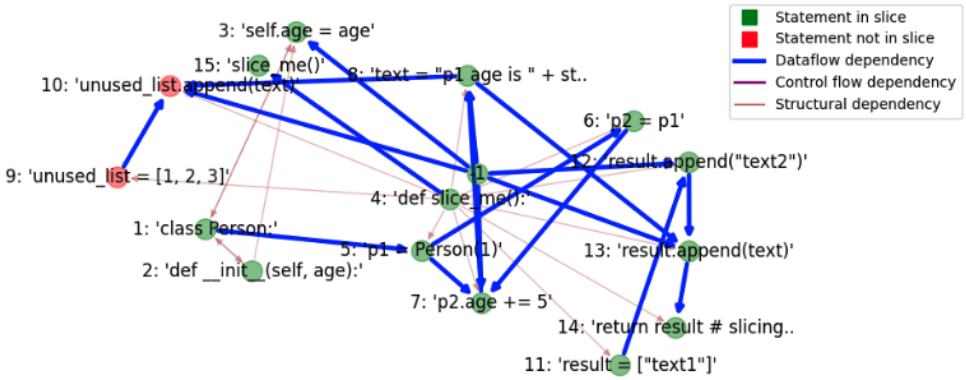

Fig. 2. Dependency graph

A.3 Scenario *Controlflow*

Code Listing 12. Source code

```
1  def slice_me():
2      a = 0
3      if a == 4:
4          a = 7
5      if a > 3:
6          a += 5
7      else:
8          a = 3
9      i = 0
10     while i < 5:
11         a += 3
12         i += 1
13     return a # slicing criterion
14 slice_me()
```

Code Listing 13. Slice

```
def slice_me():
    a = 0
    if a > 3:
        pass
    else:
        a = 3
    i = 0
    while i < 5:
        a += 3
        i += 1
    return a # slicing criterion
slice_me()
```

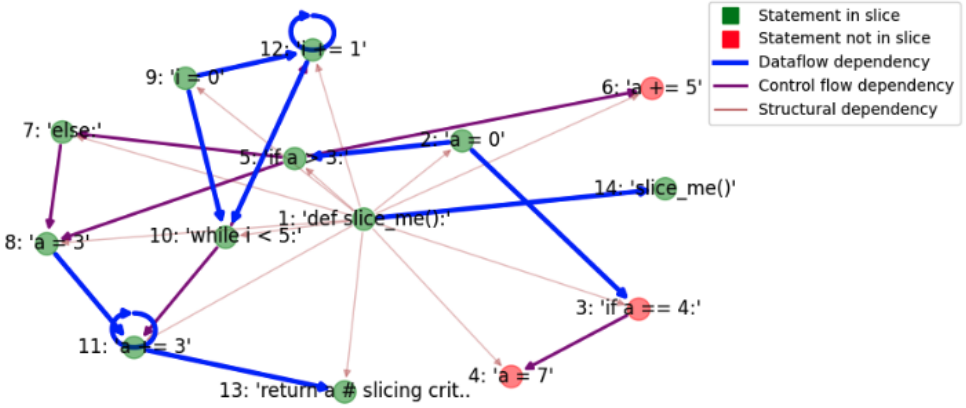

Fig. 3. Dependency graph