BackSlicer: A Lightweight Backward Slicer

Mohammed Sayagh, Bram Adams
Polytechnique Montreal, Canada
{mohammed.sayagh, bram.adams}@polymtl.ca

***Abstract*—Source code slicing allow a user to focus only a part of the program that may have a relation with a given line. From its introduction by Weiser, source code slicing is a technique that becomes popular, and which is widely used to resolve different kind of problems in software engineering. However, source code slicing is a time-consuming technique and requires an important memory size. In this paper, we present an approach to make the static slicing less time consuming, scalable, and that do not requires a huge memory size, especially in the case of big softwares. Our tool takes less than 15 minutes to perform the slicing for the whole source code of Linux instead of hours or days.**

I. Introduction

Slicing is a technique widely used for different purposes. It is an efficient manner to understand a source code or debug it. After its introduction by Weiser et al. [1], slicing is widely used in the literature to resolve different kind of problems.

Despite the popularity of slicing, performing a static slicing on a source code application is considered as a time consuming task, especially for applications that have a large number of source code lines, like Linux. Moreover, slicing is not only time-consuming, but it requires an important memory size.

To overcome this problem, an approach was proposed by Newman et al. [3], but it considers only forward slicing. However, backward slicing requires more data comparing to forward slicing. Therefore, in this paper, we present an approach to make backward slicing fast and efficient. Similar to SrcSlice [3], our tool takes as input a parsed source code by SrcML [2].

This paper is organized as follow, we present the approach is Section II, we then discuss its implementation in Section III before concluding and discussing future work in Section IV.

II. Approach

Our tool takes as input the parsed source code, which is parsed by SrcML [2].

A. General Overview

By reading the parsed source code file, we start by performing an intra-procedural slicing of the whole functions. Then, we add call graph information in order to consider inter-procedural slicing.

The key point of our slicing approach that makes it fast is that we read the parsed file only once. By reading that input file, we collect data that serves as slicing of each line in the tool. Moreover, we only collect the minimum data in order to make our solution scalable. Let's consider the example in Listing 1.

By traversing the parsed source code file (Generated by using SrcML), we generate a model based on our meta-model that is presented in Figure 1.

Listing 1: Our Runing Example

```
1  // Block 0 - level 0
2  int global_var;
3  void foo (int* x) {
4       int a;
5       *x = 19;
6       x = &a;
7       if ( global_var >10) {
8            *x = 20;
9       } else {
10           *x = 0;
11      }
12      print("%d", a);
13 }
14 int bar (int y) {
15      global_var = 20;
16      foo(&y);
17      return y;
18 }
19 int main (int argc, char** argv) {
20      // Block 1 - level 1
21      int a = 10;
22      int b = 20;
23      if (a >=20) {
24           // Block 2 - level 2
25           b = bar(a);
26           printf ("%d", b);
27      } else {
28           // Block 3 - level 2
29           b = bar(a) + 10;
30           printf ("%d", b);
31      }
32      printf ("%d", b);
33      return 0;
34 }
```

B. Data Flow

By traversing the parsed file and for each function, we collect their variables, where they are defined, and where they are used. For example, in function "main", we have the variable "a" defined in line 21, and used in lines 23 and 25. Therefore, we can say that the slicing of lines 23 and 25 is line 21.

Within the main function, we also have the variable "b", which is defined in line 22, and used in lines 26, 30, and 32. The variable "b" is also modified in lines 25 and 29, which we consider as a def of that variable as well.

Therefore, the backward slice of line 32 corresponds to the lines 25, 29,and 22.

However, for the backward slice of line 30, we should consider the execution scenario possibilities, which reflects the

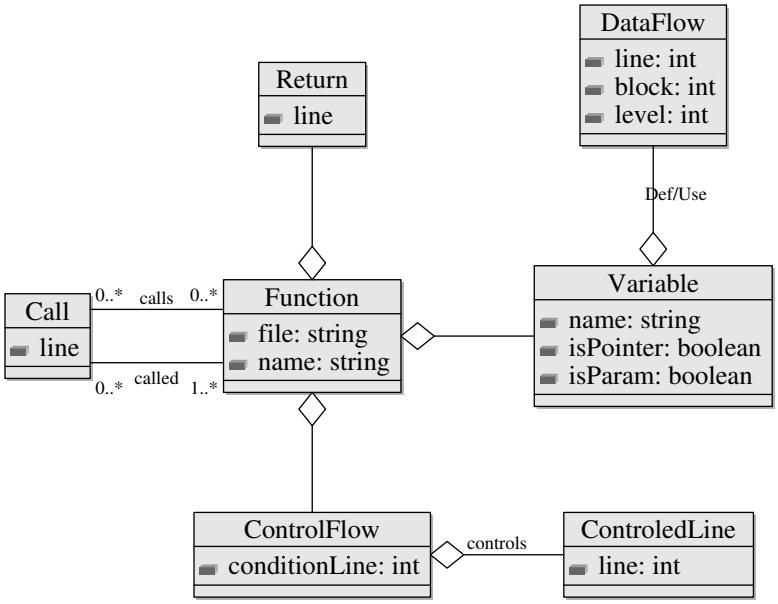

Figure 1: Meta-Model to which a source code is prepared for slicing.

impossibility to execute line 25, as it is in the true block of condition "a>= 20", whereas the sliced in line is the false branch. Therefore, we don't report that line. Moreover, we don't report line 22, as line 29 modify the value of the variable "b", and hence cancel the impact of line 22.

In order to consider the execution order, we give to each block an id and a level, which allow us to model the possible executions scenarios, and hence reduce false positives as shown in the previous example.

C. Pointers

Similar to the slicing of variables that we discussed in the previous Section, we also consider pointers. Let's consider the example of function "foo" in our running example of Listing 1. The pointer "x" is declared in line 3, and it points to the variable "a" at line 6. Therefore, each following modification of the pointer value ("*x = ...") is considered as a modification of variable "a" as well, hence the lines 8 and 10 are considered as modification not only to "x", but also to the variable "a".

By this manner, we can find that the slice of line 12 is the lines 8, 10, 4. But, by considering the possible execution scenarios we discussed in the previous Section, we can reduce the slice of line 12 to only lines 8 and 10. Note that we only discuss about data flow in this Section, we will discuss about other aspects in the following Sections.

D. Control Flow

We do not consider only data flow to build a slice, we should also consider control flow as well. We only need to find which condition is controlling which block, and hence which source code lines in that block. For example, the condition in line 23 controls blocks 2 and 3. Therefore, we should add to the backward slice of each of the lines from 24 to 30 the line 23.

The control flow information is represented by the classes "ControlFlow" and "ControledLine".

Note that we only save the line numbers, which reduces the required memory and hence improve the scalability of our approach. Moreover, we perform that control flow data collection approach in the same single traverse of the parsed source code, no need for more iterations to compute the control flow data.

E. Call Graph

During the same iteration in which we traverse the parsed source code, we collect call graph data. By parsing each function, we collect the set of methods called by that method and in which line they are called. For example, function "foo" calls "printf" at line 12, function "bar" calls the function "foo" at line 16, function "main" calls "bar" at lines 25 and 29, and it calls "printf" at lines 30 and 32.

At the end of the first iteration, we have all function calls with all the necessary information, which are the set of caller functions, called ones, and in which source code line that call happens. All we have to do then is to map these calls, by matching the caller and the called functions, and that is by using the class "Call" in Figure 1. Therefore, the slice of a line that calls another function should include the slice of that function return statements.

Therefore, we use these information to build inter-procedural backward slice for each line. An example of that inter-procedural analysis is the slice of line 29, which is in addition to the lines 23 and 21, which are collected by data and control flow analysis, we add the slice of line 17, as it is the return statement of function "bar". Recursively, we add the slice of line 17 as well to the slice of line 29.

Moreover, if a function called has a pointer or a reference variable in the parameters, we add the def of that parameter to the slice lines of the call, until changing the address of that pointer. Let's consider the example of line 16, in which the function "bar" calls the function "foo", which has the pointer "x" in the parameter. Therefore, the modification of the value of pointer "x" should be included in the slice of the call (line 16), until changing the address of that pointer in line 6. Similarly to the possible execution scenarios that depends on different source code blocks, we also consider it in that case of pointers.

Furthermore, we use call graph to find the calls to a function, in order to continue the backward slice. For example, the slice of the line 17 contains the line 14, which is the declaration of the function. We do not stop the analysis just there, but we continue by finding which functions call "bar", and precisely in which source code lines, to end up by adding the lines 25 and 29 to the slice of line 17. Moreover, we recursively add the slice of these two lines as well.

F. Global Variables

Our approach considers the global variables as well. We created an abstract function that we refer to by "_GLOBALS". By respecting the meta-model of Figure 1, the function "_GLOBALS" contains the global variables, where they are used, and where they are modified (def). An example of that case is highlighted in function "foo" that uses the global variable "global_var", which is not basically modified in the same function "foo", but in function "bar". By applying our technique, we can add the lines 2 and 15 to the slice of line 7.

III. Implementation

We implement our approach in the tool BackSlice. Our tool takes as input a parsed source code, which we can get by executing SrcML [2]. Then, it outputs for each source code line its slicing dependencies.

A part of building data flow was already implemented by SrcSlice [3], we added the consideration of execution scenarios as explained in Section II. We also implemented the rest of the approach that we discussed before.

IV. Conclusion

We propose in this paper an approach that makes slicing of C-code scalable and fast. The approach considers as input a parsed source code, and generates the slicing output by collecting all the necessary data in one and unique iteration. Our work is complementary to Newman et al. [3] work. We aim at evaluating the tool in future work, and we also plan to consider other programming aspects like function pointers in the future.

References

[1] M. Weiser, "Program slicing," in *Proc. of the 5th ICSE*, 1981, pp. 439–449.

[2] "Srcml," http://www.srcml.org/.

[3] C. D. Newman, T. Sage, M. L. Collard, H. W. Alomari, and J. I. Maletic, "srcslice: A tool for efficient static forward slicing," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 621–624. [Online]. Available: http://doi.acm.org/10.1145/2889160.2889173