
# REPORT

## MODULARIZATION USING LATTICE OF CONCEPT SLICES

## PROGRAMMING LANGUAGES

CS 306

*By:*
I.Cherish (IMT2017022) Cherish.Chowdary@iiitb.org
M.Khalid (IMT2017028) MohammadKhalid.Udayagiri@iiitb.org
S.Sanjay (IMT2017037) SanjayKumar.Reddy@iiitb.org

7th June, 2020

# Contents

# 1 Abstract

Code Quality of a program reflects how well the code reflects the design. It also refers how well the code is structured enough to maintain the certain qualities like robustness or maintainability. There are various metrics that are used to measure the quality of the code. In this Project we looked at a study which focuses on a program representation formalism called Lattice of Concept Slices and will define two modularization algorithms based on it. We also implemented a static analysis tool that computes the cyclomatic complexity of a given program. We implemented the tool in python for C language.

# 2 Introduction

Many software systems lack modularity and these systems become difficult to understand and maintain. Decomposition of such systems into a modular structure helps in better understanding and maintenance. Researchers have used source code analysis techniques like concept assignment, formal concept analysis, and program slicing for modularization of these systems. All these techniques have their own advantages and disadvantages which are discussed in further sections.

The authors of the paper Source Code Modularization Using Lattice of Concept Slices, Raihan Al-Ekram and Kostas Kontogiannis, introduced a program representation formalism called the Lattice of Concept Slices and propose a modularization technique based on it. This approach uses all three above mentioned techniques by using their strengths and overcoming disadvantages. The goal is to achieve modularization such that each module implements a single domain concept, is self-contained with minimal duplication in code. They also defined a new type of slicing based on the domain concepts and compute concept slices. Finally, they performed modularization by clustering and restructuring the lattice. In this paper we will try to reproduce their work.

# 3 Prerequisite

The authors of the paper used static code analysis techniques like concept assignment, program slicing and formal concept analysis to build Lattice of concept slices and program modularization algorithms. So we will discuss briefly about these concepts and their advantages and drawbacks.

## 3.1 Concept Assignment

As defined by Ted J.Biggerstaff, concept assignment problem is discovering of individual human oriented domain concepts and assigning them to implementation oriented counterparts for a given problem.

A domain concept is an idea or a task in the problem domain that is being implemented in the program, e.g. calculate interest, book ticket etc. One of the approach for identifying domain concept is structural analysis(based on parsing technology). The source code is parsed to match the signature of the pattern then matching lines are considered to be part of domain concept. The small or atomic concepts are recognized first and based on them larger composite domain concepts are identified.

The advantage of this technique is fragmentation is done at right level of granularity. But disadvantage is that the code segment identified as domain concepts are not self-contained and not executable independently as separate module.

## 3.2 Program Slicing

Weiser described it as an abstraction of a program based on a particular behaviour. It is an executable subset of the original program that preserves the original behaviour of the program with respect to slicing criteria for a given variable V at a given program point P $< P, V >$

First we will create Program Dependence Graph(PDG) for the program. The algorithm starts by traversing PDG from the node corresponding to program point P and then finds all nodes that has direct or indirect control or data flow dependency on this node. All visited nodes are the computed slices.

The advantages of program slicing is that the slices are self-contained and executable by themselves. But as decomposition is done on program variables instead of domain concepts,so this may result in duplication of code because of overlapping control flow. Duplication of code may also result in undesirable side effects.

In the below two figures(Figure 1, Figure 2), left side has original code and the right side contains the sliced code with respect to c.

In the Figure 1, as c computation depends on value a and it has declaration in line 3 i.e. a=20, we include this line. And also in if condition, the value of a may get updated and this if condition depends on x value. So we will include line x=10 and if condition. But the value of c, doesn't depend on y, so we will not include this line.

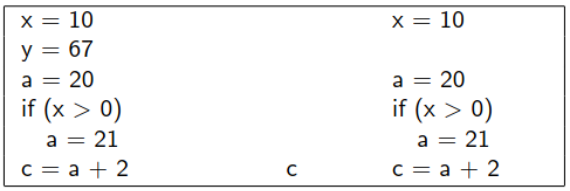


Figure 1: Slicing with respect to c

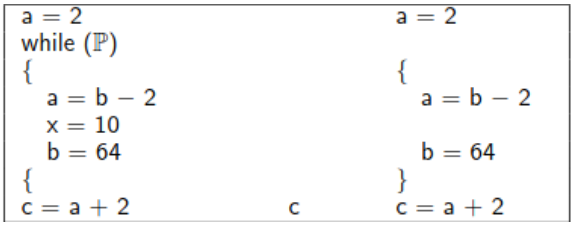


Figure 2: Slicing with respect to c

Similarly in the Figure 2, the value of c depends on a and a has declaration in first line. And also in while loop a may get updated depending on the condition P. So we should include all lines expect the line x=10(as this doesn't effect the value c). But in the code after slicing, it also doesn't contain line corresponding to while, so the corresponding slicing is wrong.

If we consider the example in Figure 3, the left side is original code and right side is slicing with respect to lines. As we can see in the slicing based on the variable lines, it contains only those statements that affect this variable. Since the else condition doesn't involve in computation of lines, it is not part of the slicing. And also the lines such as declaration of char and subtext are only helpful in calculation of characters, they are also not included in the slicing with respect to line.

## 3.3 Formal Concept Analysis

Formal concept analysis (FCA) is a principled way of deriving a concept hierarchy from a collection of objects and their properties. Each concept in the hierarchy represents the objects sharing some set of properties; and each sub-concept in the hierarchy represents a subset of the objects (as well as a superset of the properties) in the concepts above it. FCA was introduced in the 1990s by Bernhard Ganter and Rudolf Wille (1998), building on applied lattice and order theory as developed by Birkhoff and others in the 1930s. The set of objects and attributes, together with their relation to each other, form a formal context, which can be represented by a cross table, where elements on the left are objects and elements on the top are attributes.A formal Context (O,A,R) is a binary relation R between a set of objects

```
*(1)  read(text);                                (1)  read(text);
 (2)  read(n);
 (3)  lines = 1;                                 (3)  lines = 1;
 (4)  chars = 1;
 (5)  subtext = "";
 (6)  c = getChar(text);                         (6)  c = getChar(text);
 (7)  while (c != '\eof')                        (7)  while (c != '\eof')
 (8)       if (c == '\n')                        (8)       if (c == '\n')
 (9)       then lines = lines + 1;               (9)       then lines = lines + 1;
*(10)            chars = chars + 1;
 (11)      else chars = chars + 1;
 (12)           if (n != 0)
*(13)           then subtext = subtext ++ c;
 (14)                n = n - 1;
 (15)      c = getChar(text);                    (15)      c = getChar(text);
 (16) write(lines);                              (16) write(lines);
 (17) write(chars);
*(18) write(subtext);
```

(a) Example program (b) Slice with respect to $\langle 16, lines \rangle$

Figure 3: Slicing with respect to variable lines

O and their attributes A.

Let us define an operator * over a subset E of objects as set of all attributes shared

| Airlines | Latin America | Europe | Canada | Asia Pacific | Middle east | Africa | Mexico | Caribbean | USA |
|---|---|---|---|---|---|---|---|---|---|
| Air Canada | × | × | × | × | × | | × | × | × |
| Air New Zealand | | × | | × | | | | | × |
| Nippon Airways | | × | | × | | | | | × |
| Ansett Australia | | | | × | | | | | |
| Austrian Airlines | | × | × | × | × | × | | | × |

Figure 4: Example formal context airlines represented as a cross table

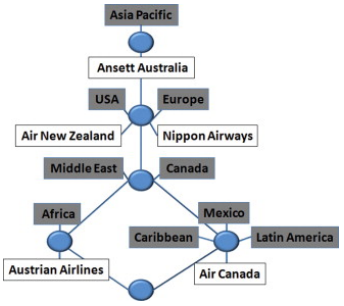


Figure 5: The Concept lattice for fromal context airlines

by all objects in E, More formally it is defined as follows.

$$E^* = \{a \in A | \forall o \in E \text{ AND } (o, a) \in R\}$$

Similarly Let us define an operator * over a subset I of attributes as a set of all objects sharing all attributes from I, More formally it is defined as follows.

$$I^* = \{o \in O | \forall a \in I \text{ AND } (o, a) \in R\}$$

A formal concept is a pair of sets (E,I), where $E \subseteq O$ and $I \subseteq A$ if and only if $E = I*$ and $I = E*$. In other words for a pair (E,I) to be a formal concept set of all attributes shared by objects in E is I and set of all objects that have attributes in I is E.

**For Example:**

- Let C(O, A, R) be a context where
- $O = \{1, 2, 3\}$ , $A = \{a, b, c\}$
- $R = \{(1, c), (1, b), (2, a), (2, b), (3, a), (3, b), (3, c)\}$
- A tuple $c_1 = (\{1, 2, 3\}, \{b\})$ is a concept, Since $\{1, 2, 3\}^* = \{b\}$ and $\{b\}^* = \{1, 2, 3\}$.

A concept $C_1(E_1, I_1)$ is sub-concept of other concept $C_2(E_2, I_2)$ if $E_1 \subseteq E_2$ or $I_2 \subseteq I_1$. The sub-concept relation forms a partial order over the set of concepts and also forms a complete lattice. The example is given in figure 5 for context in figure 4. Concepts in lattice are grouped together depending on the relationships among them. It is used in several software applications like program understanding, automatic modularization of legacy, etc.

**For Example:**

- $c_2 = (\{1, 3\}, \{b, c\})$ is a concept.
- Concept $c_2$ is a sub-concept of concept $c_1$, Since $\{1, 3\} \in \{1, 2, 3\}$.

**Proof that the relation sub-concept is a partial order**:
For any relation to be a Partial order it has to satisfy three properties:

1. Reflexive
2. Asymmetric
3. Transitive

**Reflexive:** A concept is a sub-concept of itself, because C(E,I), $E \subseteq E$.
**Asymmetric:** If $c_1$ is a sub-concept of $c_2$ and $c_2$ is a sub-concept of $c_1$, then $c_1 = c_2$. From the premise we can get the relation between the extents of the two concepts as $E_1 \subseteq E_2$ and $E_2 \subseteq E_1$, from the results in set theory we can conclude that $E_1 = E_2$. Similarly for intents we get a relation $I_2 \subseteq I_1$ and $I_1 \subseteq I_2$ and we can conclude that $I_1 = I_2$. Hence $(E_1, I_1) = (E_2, I_2)$, which gives us the same result as the conclusion of the claim.
**Transitive:** If $c_1$ is a sub-concept of $c_2$ and $c_2$ is a sub-concept of $c_3$, then $c_1$ is a sub-concept of $c_3$.

We will prove the relation between extents of the concepts this will naturally give us the relation between intents of the concepts. From the premise we can get the following relations $E_1 \subseteq E_2$ and $E_2 \subseteq E_3$, Hence we can conclude that $E_1 \subseteq E_3$, which proves the argument. Hence the sub-concept forms a partial order over the set of concepts.

# 4 Lattice of Concept Slices

Authors(Raihan Al-Ekram and Kostas Kontogiannis) proposed program representation called lattice of concept slices using concept assignment,program slicing and formal concept analysis. The formation of lattices consists of three stages.

1. Identification of Domain Concepts
2. Computation of Concept Slices
3. Building the Lattice

## 4.1 Identification of Domain Concepts

In this approach, software engineer provides a list of domain concepts that are taken form functional specifications of the system. We can also combine domain concepts with one or more program elements such as variables and structural idioms in the source code. And also for a given concept, some statements will be more important than others i.e. which will contribute more to computation of domain concept. This statements are identified as key statements.

Some criteria to select it as domain concepts are identifiers such as return statements, modified formal parameters that have been called by references, variables in output and input statements, global variables or class attributes that have been modified i.e. any information being sent outside the program or any change in internal state that is externally visible or information that is used by other parts of the program. We also identify key statements along this. The programmer has the choice to accept or reject these identifiers. Each concept is candidate to form a possible module.

```
1: #include <stdio.h>
2: #define YES 1
3: #define NO  2
4: void main()
5: {
6:   int nl = 0;
7:   int nw = 0;
8:   int nc = 0;
9:   int inword = NO;
10:  int c = getchar();
11:  while (c!=EOF)
12:  {
13:    char ch = (char) c;
14:    nc = nc + 1;
15:    if (ch=='\n')
16:      nl = nl + 1;
17:    if (ch==' ' || ch=='\n' || ch=='\t')
18:      inword = NO;
19:    else if (inword == NO)
20:    {
21:      inword = YES;
22:      nw = nw + 1;
       }
23:    c = getchar();
     }
24:  printf("%d \n", nl);
25:  printf("%d \n", nw);
26:  printf("%d \n", nc);
   }
```

Figure 6: c program to compute number of lines

**Table 1: The Domain Concepts**

| Domain Concepts | Statements |
|---|---|
| Lines | 6, **16**, 24 |
| Words | 7, **22**, 25 |
| Chars | 8, **14**, 26 |

Figure 4 contains c program to calculate number of lines, words and characters. Now we shall compute domain concepts of this. We will decide domain concept as discussed above with respect to output statements i.e. print statements. So, the domain concepts will be lines, words and chars. If we consider domain concept with respect to lines, then the statements 6,16 and 24 come under this as in line 6 it has declaration, line 16 computes the value and line 24 prints it. Similarly 7,22,25 and 8,14,26 will be statements to domain concepts words and chars respectively. And line 16, 22, 24 will be key statements of respective domain as this statements does the main computation. Table 1 describes the same(bold are the key statements).

## 4.2 Computation of Concept Slices

Concept slice is a slice of the program with respect to domain concept. It is computed by taking slices of program with respect to statements belonging to domain concept and then taking union of slices. This step add more statements to domain concept and make it executable. The concept slices corresponding to domain concepts are candidates for possible modules.

Now consider the same example from figure 2.Now we shall calculate concept slices with respect lines, words and chars. If we consider slice of the domain concept 'lines', then all statements from domain concept will include and also some additional statements related to this computation like the statements from lines

10,11,12,13,15,23 and also lines like 4,5 will also be included to make it executable independent. Figure 5 shows the same i.e. concept slice of the domain concept 'lines' and Table 2 shows the concept slices of their all the domain concepts discussed above.

```
4: void main()
5: {
6:   int nl = 0;
10:  int c = getchar();
11:  while (c!=EOF)
12:  {
13:    char ch = (char) c;
15:    if (ch=='\n')
16:      nl = nl + 1;
23:    c = getchar();
     }
24:  printf("%d \n", nl);
   }
```

Figure 7: Slice of the domain concept lines

**Table 2: The Concept Slices**

| Domain Concepts | Statements |
|---|---|
| Lines | 4, 5, 6, 10, 11, 12, 13, 15, **16**, 23, 24 |
| Words | 4, 5, 7, 9, 10, 11, 12, 13, 17, 18, 19, 20, 21, **22**, 23, 25 |
| Chars | 4, 5, 8, 10, 11, 12, **14**, 23, 26 |

### 4.3 Building the Lattice

The lattice is built by performing formal concept analysis on the concept slices from the previous phase. The context is formed by the relationship between domain concepts and program statements. Here the domain concepts corresponds to objects(Extent) and statements corresponds to attributes of context(Intent). Concept lattice is formed by performing formal context analysis on the context. A domain concept is related to the statement if that statement belongs to the concept slice for that domain concept let us denote this relation by $\sigma$. More generally, each domain concept in lattice is related to all statements in its node and all nodes below it, whereas each statement is related to all the domain concepts in its node and all the nodes above it(same as formal concept analysis as we discussed in section 3.3). Figure 7 is the resultant lattice after performing formal concept analysis on table 2.

## 5 Modularization Algorithms

Based on the representation of lattices of concepts slices,the authors proposed new modularization techniques. The goal of this is to achieve a modularization such that each module implements a single domain concept(preferably), each module is self-contained with minimal duplication of code and no side effect among modules.

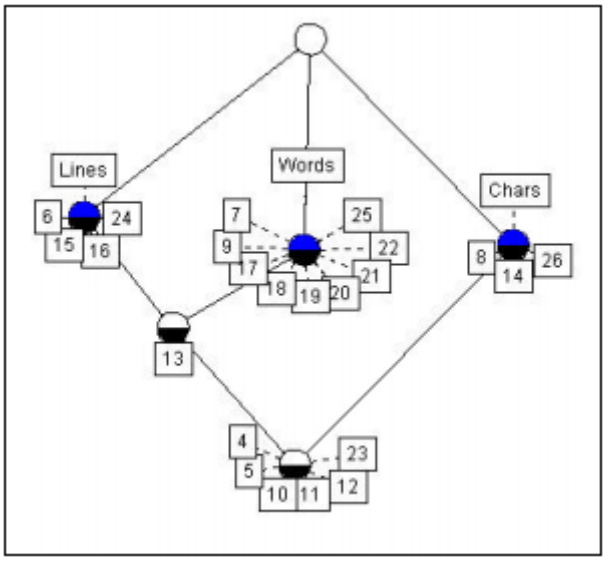


Figure 8: Lattice

## 5.1 Lattice Clustering Algorithm

To explain this algorithm, they first defined a critical node. A critical node is a node that contains a principal variable. A principal variable is a variable in a set of statements, if it is global or call-by-reference and is assigned in those statements. Since a statement is related to all the domain concepts above it, a critical node will affect all the domain concepts above it.

To solve this, the authors gave a clustering algorithm. In this algorithm, we first run a bottom-up traversal. In this traversal, if the node is a domain concept, we create a cluster with this node. If it is a critical node, then we create a cluster with this node and all the nodes above it. In figure 7, sub-figure (a), since there are no critical nodes and nodes 3, 4, 5 are domain concepts, they have separate clusters. In sub-figure (b), node 2 is a critical node and nodes 3, 4, 5 are domain concepts. So, nodes 2, 4, 5 form a cluster and node 3, a different cluster. In the next step, we merge the clusters which have an intersection. In sub-figure (c), nodes 2 and 3 are critical nodes. So nodes 2, 4, 5 form a cluster and nodes 3, 5, 6 form a cluster. These two clusters intersect at node 5. So, we merge these clusters. In the final step, we create separate modules for each of these clusters.

## 5.2 Lattice Restructuring Algorithm

In the previous algorithm code duplication might happen. To avoid this, we can restructure the lattice such that the statements will have a complete control flow.

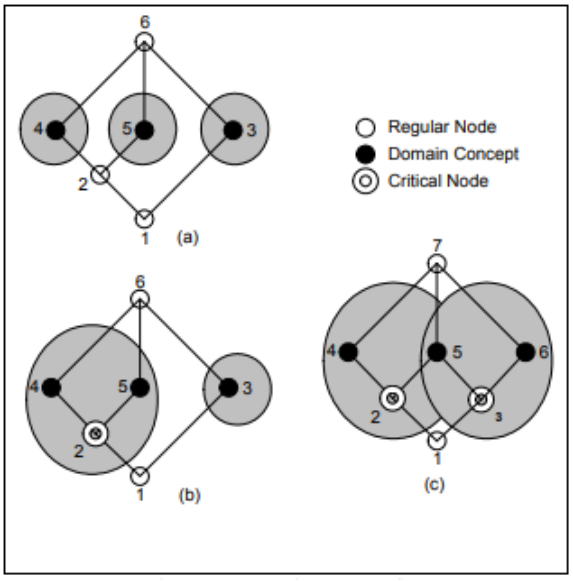


Figure 9

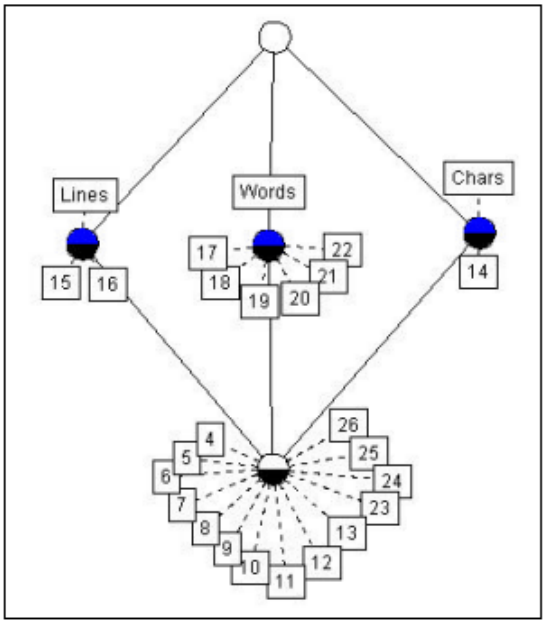


Figure 10

In the top-down traversal, we first identify all consecutive statements with a key statement as a group. If there are no consecutive statements, we take the key statement as a group. All the other statements are pushed down to the next node. In Figure 6, in the domain concept lines, statement 16 is a key statement. So, we take its consecutive statements, i.e. statements 15 and 16 as a group. Whereas in the domain concept chars, 14 is the key statement and there are no consecutive statements. So only 14 is taken as group and all other statements are pushed down. Next we create a module for each group in this node. In the bottom-up traversal, from the module of this node we call the modules of the nodes above it. So, the last node in the lattice becomes the controlling module. On applying this algorithm to the lattice in figure 6, we get the lattice in figure 8.

# 6 Building Control Flow Graph for a given C program

Out approach was to construct the CFG form the Abstract Syntax Tree that we get after parsing the C source with parse_file function which can be imported from pycparser library. After getting abstract syntax tree(AST), we traverse the AST and construct CFG as follows

- Traverse the AST.
- Extract the values from AST
- Pass the extracted values to the statement components
- Pass the statement component to the CFG node.

# 7 Cyclomatic Complexity

It is a quantitative approach to measure the structural complexity of code. It is done by counting number of linearly independent paths(the path which has at least one edge which has not been traversed before in any path) in the source code. We calculate it with the help of control flow graphs(CFG). CFG is a graphical representation of all paths that might be traversed in a program while executing. Cyclomatic complexity is calculated by the formula, cyclomatic complexity = E - V + 2, where E is the number of edges in CFG and V is the number of vertices in CFG.

Linearly independent paths in the CFG given in figure 11 are

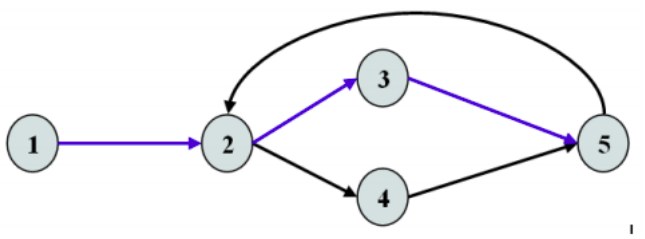


Figure 11: Example Control Flow Graph

1. 1,2,3,5
2. 1,2,4,5
3. 1,2,4,5,2,3

The number of edges in the given CFG is 6 and the number of nodes in the given CFG is 5. Hence, by applying the formula E-V+2 we get 3. Which is consistent with what we found out by manually checking. We have implemented the CFG construction for only a subset of C language

- If
- If Else

- Sequence of Statements - Declaration, Assignment
- Iteration Statements.
- Function Calls - User defined Function calls.

**Implementation:**
In order to construct a CFG for specific type of statements we used $build_block$ which checks the instance of AST node and will call the appropriate sub-routine. This function takes as input a node of AST and return start and end node of the CFG for that subtree. For example for while statement we designed the subroutine in the following way:

```
def build_block(self, asst):
    j = asst
    vertices = 0
    edges = 0
    if(isinstance(j, pycparser.c_ast.While)):
        curr = CfgNode("while "+get_source_string(j.cond))
        vertices+=1
        st1, en1, ver_res, edges_res =self.build_block(j.stmt)
        vertices += ver_res
        edges += edges_res
        curr.adj_list.append(st1)
        edges+=1
        end = en1
        for i in end:
            i.adj_list.append(curr)
            edges+=1
        end = [curr]
        return (curr, end, vertices, edges)
```

Figure 12: Code snippet for construction of while gadget

- Get the source string for the condition and store it in the starting node.
- Call the function build block on the body of while recursively
- After getting the gadget corresponding to the block add the edges between them accordingly.
- Add the edge from the end nodes of all the while body gadgets and starting node. This is because after the execution of the block the program flow goes

to the while and checks the condition.

- The start and end nodes of the while block will be returned at the end of the block.

# 8 References

[1]Keith Gallagher, David Binkley. (n.d.). Program Slicing. Retrieved from http://www.cs.loyola.edu/~binkley/papers/fosm08-slicing.pdf
[2]Mccabe's Cyclomatic Complexity: Calculate with Flow Graph (Example). (n.d.). Retrieved from
https://www.guru99.com/cyclomatic-complexity.html
[3]Raihan Al-Ekram, Kostas Kontogiannis. (n.d.). Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.137.8274&rep=rep1&type=pdf
[4]SILVA OSEP . (2008, December). An Analysis of the Current Program Slicing and Algorithmic Debugging Based Techniques. Retrieved from https://pdfs.semanticscholar.org/b963/ea7d820b66f0077809de9f06cfd50b42fc35.pdf