Technical Report

MDH-MRTC-284/2014-1-SE

Objects-Based Slicing

Husni Khanfar
husni.khanfar@mdh.se

May 2014
School of Innovation, Design and Engineering
Mälardalen University

Contents

1 Introduction **5**

2 Background **6**
2.1 While Language 6
2.2 The Syntax of the WHILE language 6
2.3 Strongly Live Variables(SLV) 7

3 Object-Based Program Representation **7**
3.1 Dividing the source code into individual Objects 8
3.2 The Interfaces 9
3.3 The Type of the Objects 9

4 Objects-Based Slicing **9**
4.1 The Slicing Criterion 10
4.2 Strong Live Variable (SLV) 10
4.3 period 10
4.4 Single-Set SLVs 10
4.5 Calculating Start and Stop Properties 11
4.6 Calculating Period From: Start,Stop, HeaderLine and LastLine Properties 13
4.7 Detecting a Data Dependency 14
4.8 Finding all the Data Dependencies inside an Object 14
4.9 Detecting the Data Dependencies between the Objects 17
4.10 Detecting the Control Dependencies 18
4.11 IF .. ELSE block 19

5 The Algorithm **22**
5.1 The Scanning Stage 22
5.2 The Main Algorithm - Slicing Program 22
5.3 The Algorithm of the Slicing Function 23

6 Experiments **25**
6.1 Number of the Variables and Number of the Iterations 25
6.2 Excluding the Parsing Time 25
6.3 Variation in the number of the lines 25
6.4 Testing the Scalability 27
6.5 Memory Saving 27

7 Results and Discussions **28**

Preface

This project is one of the part of my PhD study in the Computer Science. I would like to thank my supervisors Professor Bjorn Lisper and Dr. Masud Abu Naser for continuous encouragement and feedback. Im very thankful to Prof. Bjorn for providing concrete ideas and Dr. Masud for cooperating both in terms of time and frequency.

Husni Khanfar,
Mälardalen University, May 2014

Abstract

Object-Based Program Representation (OBPR) consists of *Objects* and *Interfaces*. Each object represents a set of statements by properties. OBPR divides the source code into objects which are connected by *Interfaces*. Every interface describes an opposite direction path of an execution flow between two different objects. OBPR does not make detailed a prior analysis to calculate the data and control dependencies like Program Dependence Graph(PDG).

Object-Based slicing is a technique for slicing well-structured codes. In this technique, the objects work individually in order to find the internal data and control dependencies. The slicing algorithm detects the internal data dependencies inside each object by implementing the Strong Live Variable dataflow analysis.

Object-Based slicing provides: a 100% slicing accuracy in comparison with PDG-based slicing. In addition, detecting the dependencies is simultaneous with the slicing process, thus the slicing is done on the fly as well as the control and data dependencies are detected on demand.

Practical experiments measure the gotten speedup from applying Object-Based Slicing. Conventional PDG-Based Slicing is used as a baseline. In terms of the total time, including parsing, we obtain speedups up to 11.6 and in average 3.98. By comparing the analysis times only after excluding the parsing times, the average time that is obtained equals 21. In some cases, the speedup reaches to about 800.

1 Introduction

Many computer science disciplines use the slicing like software debugging, software testing, software measurement, program comprehension, software maintenance and program parallelization. *Slicing* extracts the statements that influence the value of a variable in a particular program location.[1, 15].

Program Dependence Graph (PDG) and Systems Dependence Graph (SDG)[6] are the main program representations used by the major slicing algorithms since 1985[2]. In PDG-Based Slicing, all the control and data dependencies of the source code are concluded and determined beforehand regardless of whether they are needed by the slicing algorithm. Thus, the amount of the computations is mainly proportional to the number of the statements and not to the dependencies that are checked by the slicing algorithm

For instance, assume the outcome of slicing 10,000 statements according to a specific slicing criterion is only two consecutive statements. Apparently, there is only one data dependency that is used by the slicing algorithm. The problem appears when we know that all the data and control dependencies must be determined and calculated in order to build the PDG of this 10,000 statements. Thousands of dependencies relations must be found in order to use eventually one of them. Consequently, the vast majority of the computations and resources, which are consumed in determining these dependencies, are useless.

Practically, large industrial software systems may consist of millions of lines. When one of PDG slicing algorithms is used to represent one of these enormous industrial systems, then extraordinary number of data and control dependencies should be calculated by a very detailed analysis. Hence, a massive task takes place.

This paper proposes a new slicing method for well-structured codes. This method accurately detects the control and data dependencies on demand during the analysis and get rid of finding beforehand all the dependencies. The new proposed approach is defined *Object-Based Slicing*

In Object-Based slicing, the source code is represented by *Objects* as well as *Interfaces* . Within this description, *Block* refers to a condition and a set of statements that are immediately dominated by this condition. For example, `while` and `if` blocks. In Object-based Program Representation, each Block is modeled by an Object. Interface refers to any two consecutive statement belonging to two different objects. The primary feature of the Object is that it collects a set of sequenced statements, ordered in accordance to the flow of the executions. In other words, every statement inside the Object is considered as an immediate predecessor to the statement that is after it in the Object. In OBPR, the statement does not have more than one predecessor that belongs to the its object. This sequenced statements set is defined as unique-predecessor environment.

In the abstract level of Objects-based program representation; the source code is divided into sequenced sets, which are connected by unidirectional Interfaces. Each Object is considered as an independent entity that is in charge of finding its internal data and control dependencies.

In contrast to the unique-predecessor environment, PDG is used to handle the fact that each statement may have many predecessors and many successors. This environment can be defined as many-predecessors. Changing the environment and the program representation will indeed change the slicing algorithm, which in its turn may increase or decrease the performance of the slicing process. Hence, we are not talking here about two different slicing methods, but we are talking also about two different program representations.

For detecting the data dependencies, a new light-weight method is improved to suit the unique-predecessor environment existing in Object. The method is defined as Single-Set Strong Live Variable (SLV) analysis[4]. From its name; this method is derived from the standard SLV dataflow analysis[4] but differs from Standard SLV analysis in the way of storing the SLVs. Single-set SLV analysis collects the SLVs of every Object in a single-set. Conversely, standard SLV analysis stores the SLVs in the program points. On other side, Single-Set SLV analysis inherits from the Standard SLV analysis the ability to start the analysis from a particular location with a specific subset of variables.

Finally, Objects-Based Slicing provides us with the following interesting contributions:

1. Use of SLV analysis avoids building the data dependency graph (DDG), doing the slicing on the fly.

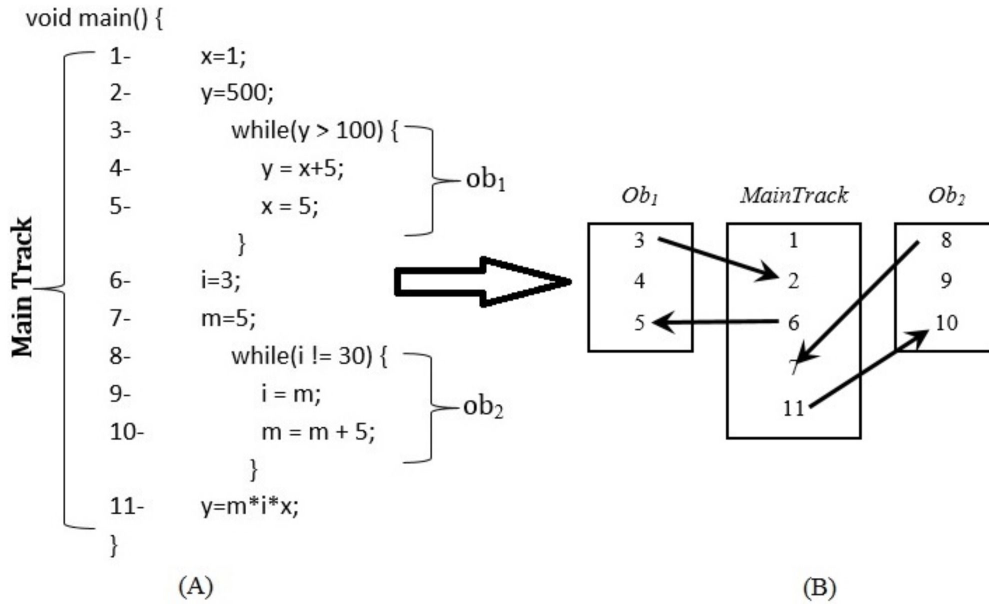

Figure 1:

2. Capturing the control dependencies in the object-based program representation avoids having to build the control dependence graph (CDG).
3. A technique to avoid having a set of SLV's in each program point, using a single SLV for each block, which also avoids the extra iterations to check for convergence done by a traditional fixed-point iteration.
4. The control and data dependencies are detected on demand.

Fig.1(A) exemplifies the concept of *Objects* and *Interfaces*. Fig.1(A) shows three objects: MainTrack object, which has the lines {1,2,6,7,11}. ob_1 object has the lines{3,4,5} and ob_2 has the lines {8,9,10}. In this example, we can consider ob_1 and ob_2 child blocks in MainTrack block.

There are four interfaces, that connect the individual objects:
$\{(11, 10), (8, 7), (6, 5), (3, 2)\}$.

Figure 1(B) shows the abstract level of the code in Fig.1(A).

2 Background

2.1 While Language

The language that is used in the slicing is WHILE language, which is a simple model language. WHILE lacks some features like pointers, arrays, unrestricted GOTO, and function calls.

2.2 The Syntax of the WHILE language

[1] The following are the syntactic categories in WHILE language:
a ∈ AExp: AExp is the set of the arithmetic expressions
b ∈ BExp: BExp is the set of the boolean expressions
S ∈ Stms: Stms is the set of all statements
We assume some countable set of variables is given; numerals and labels will not be further defined and neither will the operators:

x,y ∈ Var	Varaibles
n ∈ Num	Numerals

[1]The material of this subsection is pasted from: F. Nielson, H. Nielson, C. Hankin, Principles of Program Analysis, Springer

$\ell \in$ Lab Labels
$op_a \in OP_a$ Artihmetic Operators
$op_b \in OP_b$ Boolean Operators
$op_r \in OP_r$ Relational Operators

The syntax of the language is given by the following abstract syntax:

a ::= x | n | a_1 op_a a_2
b ::= true | false | not b | b_1 $op_b b_2$| a_1 op_r a_2
S ::= $[x{:=}a]^\ell$ | $[\text{Skip}]^\ell$ | s_1; S_2 |
if $[b]^\ell$ then S1 else S2 |
while $[b]^\ell$ do S

2.3 Strongly Live Variables(SLV)

Dataflow analysis technique calculates on each program point the variables that may reference other SLVs. kill/gen functions of SLV analysis are:

$$\begin{aligned} kill([x := a]^\ell) &= \{x\} \\ kill([skip]^\ell) &= \phi \\ kill([b]^\ell) &= \phi \\ gen([x\ :=\ a]^\ell) &= FV(a) \\ gen([skip]^\ell) &= \phi \\ gen([b]^\ell) &= FV(b) \end{aligned} \tag{1}$$

FV(x) denotes the variables used on expression a.

SLV-dataflow equations are:

$$\begin{aligned} SLV_{entry}(\ell) =& (SLV_{exit}(\ell)\backslash kill_{SLV}(B^\ell) \cup \{gen_{SLV}(B^\ell)\} \\ & where(B^\ell \in STATEMENTS(S^*) \wedge kill_{SLV}(B^\ell) \subseteq SLV_{exit}(B^\ell) \end{aligned} \tag{2}$$

$$\begin{aligned} SLV_{entry}(\ell) =& SLV_{exit}(\ell) \\ & where(B^\ell \in STATEMENTS(S^*) \wedge kill_{SLV}(B^\ell) \nsubseteq SLV_{exit}(B^\ell) \end{aligned} \tag{3}$$

$$SLV_{exit}(\ell) = \begin{cases} \phi & if \ell \in final(S^*) \\ \bigcup\{SLV_{entry}(\ell')|flow(\ell',\ell) \in Flow^R(S^*)\} & otherwise \end{cases} \tag{4}$$

The reference of these equations and functions are: [4]

3 Object-Based Program Representation

An Object-Based Program Representation (OBPR) consists essentially of *Objects* and *Interfaces.* Each Object represents a basic block consisting of a set of consecutive statements and directly controlled by a single control predicate. The **while** and **if** blocks of C language are examples of basic blocks and hence objects.

In case of nested control predicates, the inner block forms the child object and the outer block forms the parent object where any statement in the child object does not belong to the parent object.

OBPR always considers the statements of the object as consecutive statements, in terms of the flow execution, even though the object contains a child object.

The Interface is a pair of two statements (ℓ_2, ℓ_1) which connects two objects obj_1 and obj_2 such that ℓ_1 belongs to obj_1 and ℓ_2 belongs to obj_2. The interface says that there is a direct flow direction from ℓ_1 to ℓ_2. In other words, ℓ_1 may be executed immediately before ℓ_2

OBPR overcomes the Program Dependence Graphs in that it does not need to calculate and conclude all the control and data dependencies in advance. In OBPR, the data dependencies are hidden inside the objects and they are concluded on demand. The control dependencies are also concluded on demands due to the fact that each object knows its parent object.

3.1 Dividing the source code into individual Objects

In the Object-Based Program Representation(OBPR), the source code is divided into individual blocks.
Each Object represents a basic block consisting of a set of consecutive statements and controlled by a single control predicate. The **while** and **if** blocks of C language are examples of basic blocks and hence objects.
In addition, the body of any Function is considered also as a block.

Each Block is represented by an Object. The Object describes and encapsulates the properties of its corresponding Block. The following list has some properties used by Objects-Based Slicing:

1. *Header_Line*: The label of the Header Line
2. *Last_Line*: The label of the Last Line
3. *ID*: The unique ID of the block
4. *Parent_ID*: The unique ID of the parent block
5. *Statement_Set*: this property stores all the statements of the block, which are controlled directly by the header of the block.
6. *Single_Set*: will be explained later
7. *Begin_Scope*: determines the start of the block by a left curly brace '{' or the statement 'Begin'
8. *End_Scope*: determines the end of the block by a right curly brace '}' or the statement 'End'

The factor which determines whether this statement belongs to the block is: whether the header of the block controls directly the execution of the statement. Hence, the statements of the child block belong to the child block and not to the parent block.

Sequenced Set of Statements: The Statement_Set property in the object is a sequenced set because the statements in it are ordered according to their execution flows. Each statement in the Sequenced Set is considered as a predecessor to the statement that is after it in the set. The order of the statements in the sequenced set is highly important.

Example 1. The source code in Fig.1(A) has three objects: ob1, ob2 and MainTrack. The propreties of these three object are:

1. ob1: *Header_Line*=3, *Last_Line*=5, *ID*= ob1, *Parent_ID*=MainTrack, *Statement_Set* = $[3, 4, 5]$
2. ob2: *Header_Line*=8, *Last_Line*=10, *ID*= ob2, *Parent_ID*=MainTrack, *Statement_Set* = $[8, 9, 10]$
3. MainTrack: *Header_Line*=1, *Last_Line*=11, *ID*= MainTrack, *Parent_ID*=NULL, *Statement_Set* = $[1, 2, 6, 7, 11]$

because IF_ELSE block has two branches, then it has additional properties:

- Instead of Statement_Set, IF_ELSE has two set of statements: Statement_Set_IF, Statement_Set_ELSE.
- Two header lines: *Header_Line_IF*, *Header_Line_ELSE*
- *Else_Statement*, *Begin_Else_Scope*, *End_Else_Scope*
- Two last lines: *Last_Line_IF* and *Last_Line_Else.*

Example 2. Fig.2 is an example of modeling an IF_ELSE block by an object. In this figure, IF_ELSE object has two statement_set: statement_set_IF=[3,4,5]. and statement_set_ELSE=[3,7,8].

```
void main() {
1-  x=1;
2-  y=500;
3-      if(y > 100) {
4-          y = x+5;
5-          x = 5;
    }
6-  else{
7-          i = m;
8-          m = m + 5;
    }
9-  y=m*i*x;
}
```

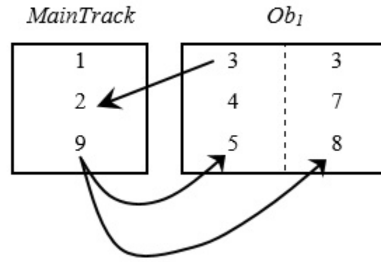

Figure 2:

3.2 The Interfaces

If there is a direct execution flow from ℓ_1 to ℓ_2 and these two statements belong to two different objects, then the The Interface (ℓ_2,ℓ_1) is created. Notice, that the interface describes the reverse direction of the flow.

Because there is a flow from the last statement in the **while** loop to the header of the **while** loop. and then from the header of the **while** loop to the immediate successor statement of the **while** loop, then we can create an interface from the immediate successor to the last statement directly. This is because **while** header does not kill any SLV according to Eq.1. In Fig.1, there is a direct flow from stm5 to stm3 and then from stm3 to stm6. but in OBPR, the interface is created in this case from stm6 to stm5 directly.

Example 3. In fig. 1, there are four interfaces:

(Stm3,Stm2), (Stm6,Stm5), (Stm8,Stm7) and (Stm11,Stm10)

According to the above interfaces, Fig.1(B) is a complete Object-Based program representation for the source code in Fig. 1(A)

Because IF_ELSE block has two last lines: *Last_Line_IF* and *Last_Line_Else*. Therefore, there are two interfaces which are created from MainTrack to obj_1 in Fig. 2, the first one is from statement 9 to statement 5 and the second one is from statement 9 to statement 8.

3.3 The Type of the Objects

There are mainly two types of Objects:

Linear Object The flow is only consecutively from the first statement to the last statement. there is no any flow from the last statement to the beginning again. The standard blocks like **if** and **if else** are examples of this kind of object.

Circular Object There is a consecutive flow from the first statement to the last. and then, there is a flow from the last statement to the first statement. This indeed causes a circular aspect of this kind of objects. The standard block **while** and **for** in C language are examples of the circular blocks.

4 Objects-Based Slicing

Objects-Based Slicing consists essentially of an OBPR (From Claim1) and an Object-Based slicing algorithm designed to work on the OBPR.

The Object-Based slicing algorithm is designed to find the internal data dependencies of each object. The Object-Based slicing algorithm implements the standard Strong Live

Variable (SLV)[4] dataflow analysis in order to find the data dependencies. Implementing SLV in Objects-Based Slicing varies than Standard SLV analysis by the following points:-

1. Standard SLV dataflow analysis stores the SLVs in the program points[4]. In Objects-Based Slicing; the SLVs of each object are collected together in a one set called Single-Set. In order to know the place of each SLV, each one is stored with associated properties.
2. Objects-Based Slicing exploits the *gen* and *kill* functions of the SLV dataflow analysis in slicing the statements, as soon as one of the statements kills one of the SLVs, then this statement is sliced.
3. in Standard SLV analysis, the SLV in a specific entry statement point is copied to the exit points of the predecessor statements in order to check later whether the predecessor statements kills the SLV. Objects-Based Slicing distinguishes between two types of the predecessor statements:
   - The predecessor statement which belongs to the same object, in this case, the slicing algorithm visits directly the predecessor statement.
   - The predecessors which belong to other objects, in this case, the SLV is reproduced in the object which the predecessor belongs to.

4.1 The Slicing Criterion

The slicing criterion is a pair $< x; n >$, where x is a variable and n is a label of a program statement. The function of the slicing is to find all the statements in the source code that may influence the value of x at n. In other words, we need to find all the statements that the slicing criterion is a data or control dependent on. In the slicing applications, we may have many slicing critera. SLV are generated and killed by the Equations in Eq.1

4.2 Strong Live Variable (SLV)

if we have the following statement:

```
100:  x=a+b
```

if x variable influences the slicing criterion, then the slicing program starts searching for all the statements which influence the value of `a` and `b`. However, `a` and `b` are not considered as new slicing criterea $< a, 100 >$ and $< b, 100 >$, they are considered as new Strong Live Variables. Notice here that the slicing criteria are converted to SLVs.

4.3 period

Period: if we assume that we have a slicing criterion or an SLV defined by the pair $< x, n >$, where x is the variable and n is the statement. n belongs to the object obj_1. Then the set of the statements in *obj1* that may influence the value of x at n is called *period.* The statements in the period are ordered in ascending manner according to its distant from n.

The distant from n = How many statements are executed till executing n.

Hence, the statements in the period are ordered from the aftermost to furthermost.

Example 4. Figures [3,4,5] are examples to the period definition.

4.4 Single-Set SLVs

Every Object has a *Single_Set* property. *Single_Set* stores all the slicing criteria and the SLVs inside it. In the *Single_Set*, each SLV is stored with its properties; *start* and *stop.* Thus, the items are stored in a triple format:

$$obj_i.Single_Set = \{(slv_1, start_1, stop_1), .., (slv_n, start_n, stop_n)\} \quad (5)$$

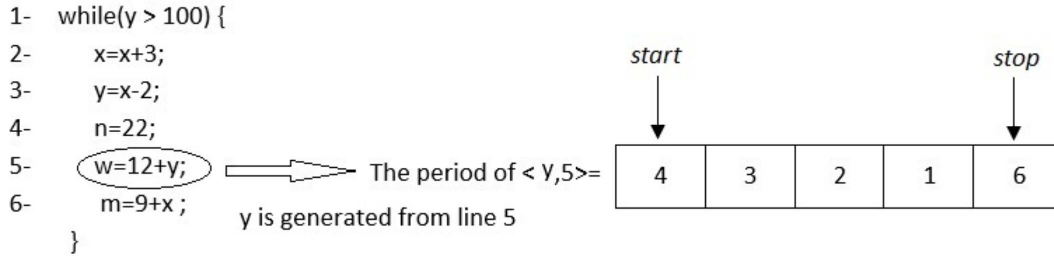

Figure 3:

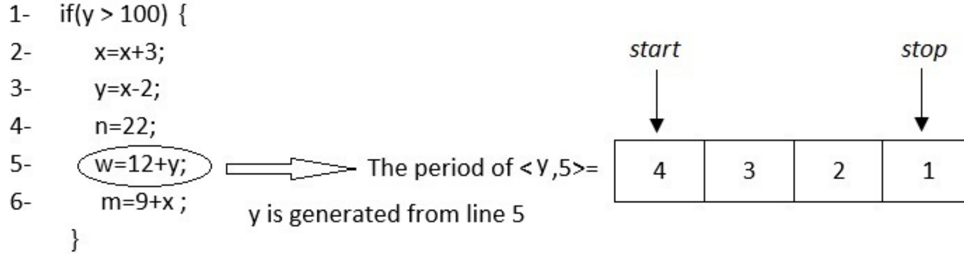

Figure 4:

start property: if $\ell_k \in$ obj.Statement_Set, then in obj.Statement_Set: The aftermost statement to ℓ_k that may influence the value of ℓ_k is: $start(\ell_k, obj)$

stop property: if $\ell_k \in$ obj.Statement_Set, then in obj.Statement_Set: The furthermost statement to ℓ_k that may influence the value of ℓ_k is: $stop(\ell_k, obj)$

Example 5. In Fig. 6(A), there are two objects; MainTrack, which its Statement_set=[1,2,3,4,5,6] and Obj1 which its Statement_Set=[7,8,9,10,11,12,13]. if the user inserts two slicing criteria: $< x, 11 >$ and $< m, 13 >$. Because the statements 11 and 13 belong to obj1, then these two slicing criteria will be created in obj1.Single_Set. obj1.Single_Set={(x,11,12),(k,13,7)}.

However, the coming section explains in depth and more how to calculate start and stop properties:

4.5 Calculating Start and Stop Properties

In order to describe the period of the SLV, two properties associate with each SLV *start* and *stop*. The value of *start* property depends mainly on:

1. Is the SLV internally generated or externally inserted.
2. Where the SLV is generated or inserted.

The value of *stop* property depends on:

1. The type of the object: circular or linear
2. the boundaries of the object: the label of the header line and the label of the last line.

Standard SLV dataflow analysis considers that the statements are represented by nodes. Every node has two points, entry point which is immediately before the node and the exit point which is immediately after the node. The SLVs are stored in these nodes. In this description:

- When we need to say that the the SLV is stored at the exit point of the statement ℓ, we say that **SLV is inserted** at ℓ. In this case, the statement ℓ becomes the first line in the period that may affect the value of the SLV. The SLVs are inserted when they are reproduced from other objects or when the slicing criterion becomes an SLV.

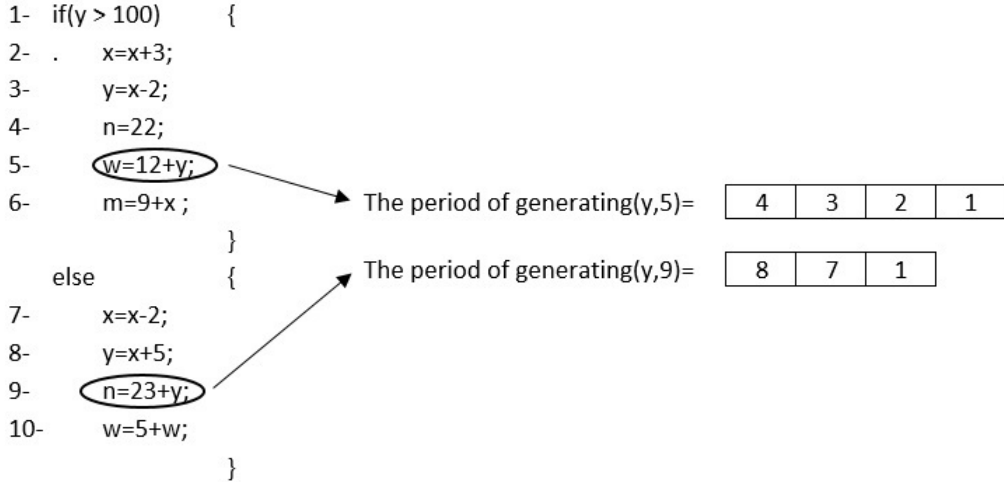

Figure 5:

- When we need to say that the SLV is stored at the entry point of the statement ℓ, we say that **the SLV is generated** from ℓ. The SLVs are generated from the statement ℓ by *gen* functions when ℓ is sliced because it has killed one of the previous SLVs. *gen* function are mentioned in Eq.1

if s_3 is a linear sequenced statements. $s_3 = [\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6]$. The variable (x) is inserted as an SLV in ℓ_4. The closest line to ℓ_4 is itself and the farthest is ℓ_1, thereupon $start = \ell_4$ and $stop = \ell_1$. $period(x, \ell_4) = [\ell_4, \ell_3, \ell_2, \ell_1]$. Notice here that ℓ_4 is the first line in *period* because itself may make an impact on the value of x at ℓ_4.

Calculating *start* property for linear and circular objects in the Insertion case: Eq.6 considers *start* property = the same line where the SLV is externally inserted.

stop: Linear object, Insertion and Generation: in the linear objects, the farthest line from any line - where SLV is either inserted or generated - is the header line. This what is reported in Eq.7 and Eq.12

In the circular object; any statement may influence directly on any another statement. Since s_4 is a circular object and $s_4 = [\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6]$, any SLV is inserted in ℓ_4 has the period : $[\ell_4, \ell_3, \ell_2, \ell_1, \ell_6, \ell_5]$. Notice here that the statements are ordered in the period in ascending with respect the distant between every statement and ℓ_4.

In the following descriptions and examples, we assume that we have a set of statements belongs to one of the objects, and the SLV is generated or inserted in ℓ_k

stop: Circular object, Insertion and Generation: Eq.8 determines the value of *stop* in two different cases;

- When $\ell_k < Lastline$: then $stop = \ell_{k+1}$ because in this case: ℓ_{k+1} is the farthest statement to ℓ_k in the object
- When $\ell_k = LastLine$: then ℓ_{k+1} does not belong to object. in this case, $stop = HeaderLine$ because it is the farthest element to ℓ_k

start:Circular and Linear Objects, Generation: Eq.9 and Eq. 11 shows that the *start* property excludes always (ℓ_k).

The following equations show how the *start* and *stop* properties are calculated for different type of objects (circular or linear) and for different kind of creating SLV (externally inserted or internally generated). In these equations consider the following:

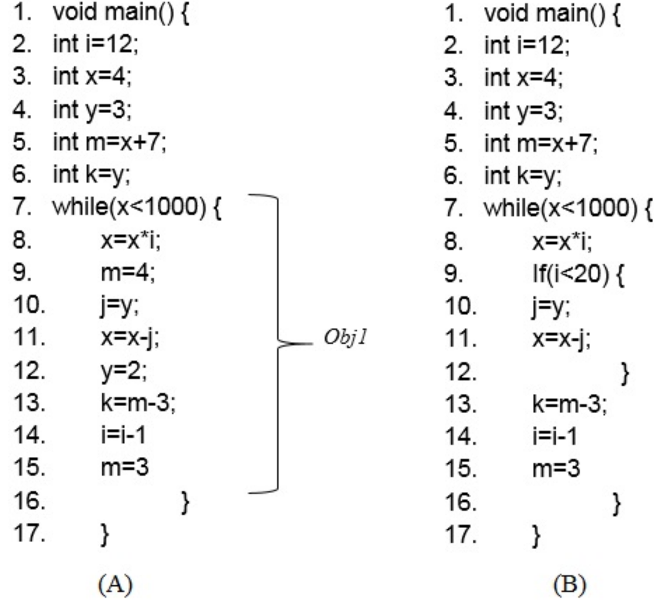

Figure 6:

$$Start_Ins(\ell_k, obj_i) = \ell_k \tag{6}$$

$$Stop_Ins_Linear(\ell_k, obj_i) = obj_i.HeaderLine \qquad if \ell_k \geq obj_i.HeaderLine \tag{7}$$

$$Stop_Ins_Circular(\ell_k, obj_i) = \begin{cases} \ell_{k+1} & if \ell_k < obj_i.LastLine \\ obj_i.HeaderLine & if \ell_k = obj_i.LastLine \end{cases} \tag{8}$$

$$Start_Gen_Circular(\ell_k, obj_i) = \begin{cases} \ell_{k-1} & if\ \ell_k > obj_i.HeaderLine \\ obj_i.LastLine & if \ell = obj_i.HeaderLine \end{cases} \tag{9}$$

$$Stop_Gen_Circular(\ell_k, obj_i) = \begin{cases} \ell_{k+1} & if \ell_k < obj_i.LastLine \\ obj_i.HeaderLine & if \ell_k = obj_i.LastLine \end{cases} \tag{10}$$

$$Start_Gen_Linear(\ell_k, obj_i) = \begin{cases} \ell_{k-1} & if \ell_k > obj_i.HeaderLine \\ \phi & if \ell_k = obj_i.HeaderLine \end{cases} \tag{11}$$

$$Stop_Gen_Linear(\ell_k, obj_i) = obj_i.HeaderLine \qquad if \ell_k \geq obj_i.HeaderLine \tag{12}$$

4.6 Calculating Period From: Start,Stop, HeaderLine and LastLine Properties

The period set of the SLVs is calculated according to the following equations:

$$\begin{aligned} period_initial(slv, obj_k) =& [slv.start, slv.start - 1, ..., slv.stop + 1, slv.stop] \\ & if\ slv.start \geq slv.stop \end{aligned} \tag{13}$$

$$\begin{aligned} period_initial(slv, obj_k) =& [slv.start, slv.start - 1, ..., obj_k.HeaderLine + 1, obj_k.HeaderLine, \\ & obj_k.LastLine, obj_k.LastLine - 1, ..., slv.stop + 1, slv.stop] \\ & if\ slv.start < slv.stop \end{aligned} \tag{14}$$

$$\begin{aligned} period(slv, obj_k) =& period_initial(slv, obj_k)[i] \mid period_initial(slv, obj_k)[i] \in obj_k \\ & if\ 1 <= i <= length(period_initial(slv, obj_k) \end{aligned} \tag{15}$$

Example 6. In Fig.6(A) and in accordance to Eq.15: period({x,11,12},obj1)={11,10,9,8,7,15,14,13,12}. period({k,13,14},obj1)={13,12,11,10,9,8,7,15,14}

When the object has a child object, then all the statements that belong to the child object must be excluded from the period. Study the following example:

Example 7. In Fig.6(B) and in accordance to Eq.15: period({k,13,14},obj1)=[13,8,7,15,14]

4.7 Detecting a Data Dependency

In the source code, every statement may have many immediate predecessors or many immediate successors in terms of the execution flow. However, the Statement_Set of the object has maximally one immediate predecessor for every statement storing in it. Similarly, it does not contain more than one immediate successor for every statement. OBPR consider the Statetment_Set of the object as a sequenced set of statements, where every statement is a predecessor to the statement existing after it in the set.
if slv is generated from the line ℓ_k. $\ell_m, \ell_k \in obj_1.Statement_Set$. ℓ_m has the index m in $period(slv, obj_1)$ ℓ_k is a data dependent on ℓ_m if the following conditions are satisfied:

- $\ell_m \in period(slv, obj1)$
- ℓ_m kills $slv.var$
- There is no statement, which exists from $period(slv, obj_1)[1]$ to $period(slv, obj_1)[m-1]$, kills slv.

According to these three conditions, Eq.16 is formed:

$$\begin{aligned} DataDependentOn(slv, obj_i) = & period(slv, obj_i)[m] \mid kill(period(slv, obj_i)[m]) = slv.var \wedge \\ & (\neg\exists period(slv, obj_i)[j] \mid j < m \wedge (kill(period[j]) = slv_i.var)) \\ & if\ j\&m\ are\ indexes\ in\ period(slv, obj_i) \end{aligned} \tag{16}$$

Example 8. In Fig.7; $obj1.Statement_Set$= $[8, 9, 10, 11, 12, 13, 14, 15]$. if we assume that we have the following slicing criterion: $< r, 13 >$. From Eq.[6,8]: $obj1.Single_Set = [(r, 13, 14)]$. period((r,13,14),obj1)= [13,12,11,10,9,8,7,15,14] . Hence, $DD(\{r, 13, 14\}, obj1) = period[5] = 9$. 5 is the index of statement 9 in period((r,13,14),obj1). Statement 13 is a data dependent on statement 9, for the following reasons:

- It assigns a value to r. Thus, the condition $kill(period[5]) = r$ is satisfied.
- Line 9 is one of the items in period((r,13,14),obj1)
- There is no statement from period[1] to period[4] assign a value to r. Thus, the condition $(\neg\exists period[j] \mid j < 5 \wedge (kill(period[j]) = r))$ is satisfied.

4.8 Finding all the Data Dependencies inside an Object

The data dependencies are detected inside the object by a special function called *Slicing Function*. Each Object has its own Slicing Function. The data dependencies are detected by the following steps:

1. Fetching individually the SLVs from the Single-Set SLV .
2. Creating a period for every fetched SLV
3. The Slicing Funciton visits the statements of the period in ascending according to the indexes . At every statement, the Slicing Function checks whether the current SLV is defined in this statement. if the current visited statement defines the current SLV, then:
   (a) if the current statement is not sliced before:
      i. the slicing function slices it
      ii. All the current variables in the right hand side of the statement are generated as SLVs and are stored in a triple format in the Single-Set SLV after calculating its *start* and *stop* properties. in this case, the new SLVs are: *generated*
   (b) The current SLV is killed

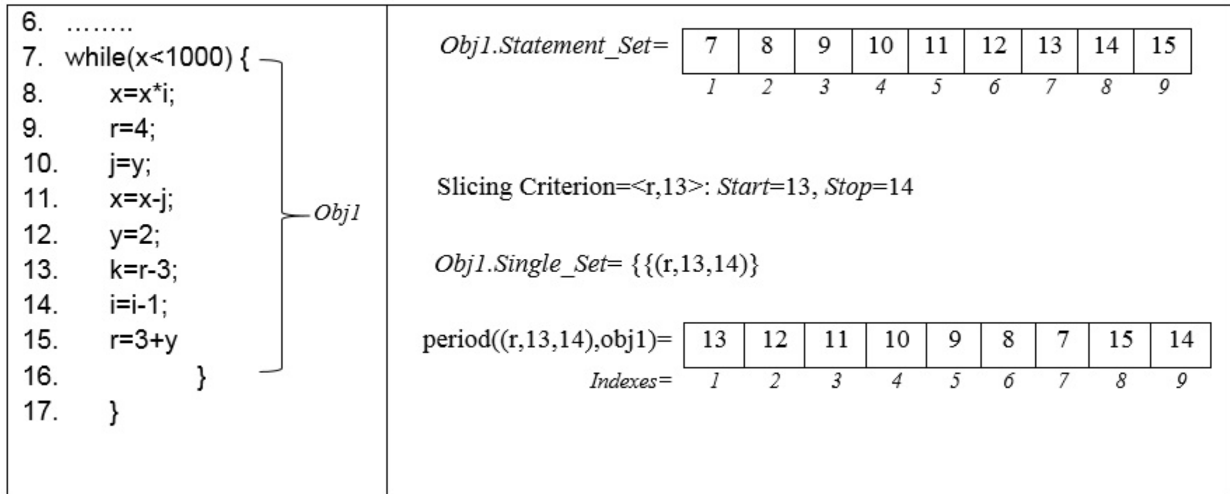

Figure 7:

(c) The slicing function stops checking other statements in the period set and returning back to the first item in this list.

4. if all the statements are visited and no statement in the period defines the current SLV, then the current SLV is killed by removing it from the Single-Set SLV.

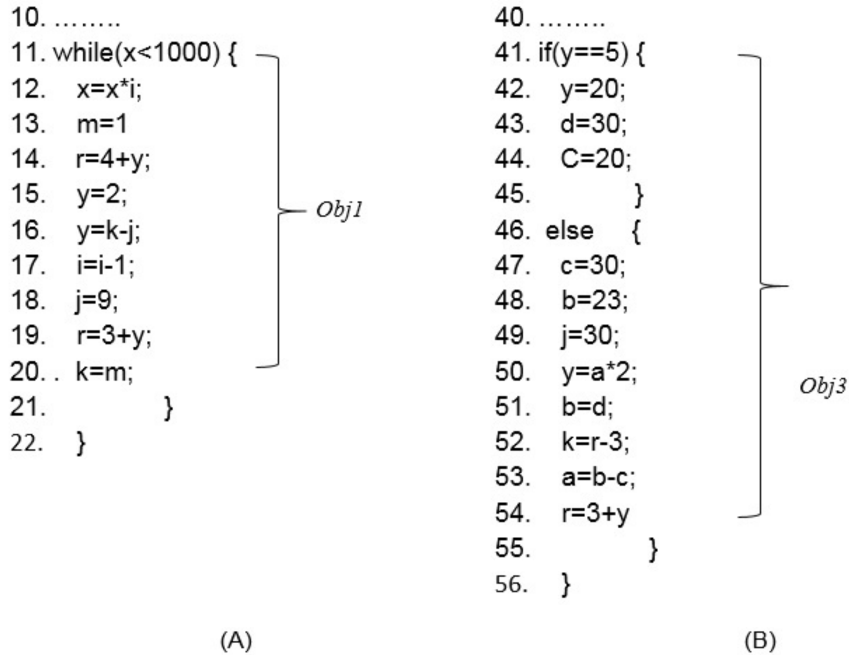

Figure 8:

Example 9 (Fig.8(A)). obj1 is a circular object. if we assume that we have the slicing criterion: $< r, 19 >$. Then the Slicing Function do the following:

- obj1.Statement_Set=[11,12,13,14,15,16,17,18,19,20].
- Slicing Criterion=$< r, 19 >\rightarrow$ From Eq.[6,8], $start = 19, stop = 20 \rightarrow$ Single_Set = {(r,19,20)}.
- Slicing_Function fetches $(r, 19, 20)$
  1. From Eq.15, period((r,19,20),obj1)=[19,18,17,16,15,14,13,12,11,20]

2. DataDependentOn((r,19,20),obj1)=19 $\rightarrow$ Sliced_Lines={19}
3. (r,19,20) is killed $\rightarrow$ obj1.Single_Set=ϕ
4. From Eq.[1], gen(19)={y}
5. From Eq.[9,10], Start_Gen_Circular(19,obj1)=18, Stop_Gen_Circular(19,obj1)=20 $\rightarrow$ obj1.Single_Set= {(y,18,20)}.

- Slicing_Function fetches $(y, 18, 20)$
  1. From Eq.15, period((y,18,20),obj1)=[18,17,16,15,14,13,12,11,20]
  2. DataDependentOn((y,18,20),obj1)=16 $\rightarrow$ Sliced_Lines={19,16}
  3. (r,18,20) is killed $\rightarrow$ obj1.Single_Set=ϕ
  4. From Eq.[1], gen(16)={k,j}
  5. From Eq.[9,10], Start_Gen_Circular(16,obj1)=15, Stop_Gen_Circular(16,obj1)=17 $\rightarrow$ obj1.Single_Set= {(k,15,17),(j,15,17)}.
- Slicing_Function fetches $(j, 15, 17)$
  1. period((j,15,17),obj1)=[15,14,13,12,11,20,19,18,17]
  2. DataDependentOn((j,15,17),obj1)=18 $\rightarrow$ Sliced_Lines={19,16,18}
  3. (j,15,17) is killed $\rightarrow$ obj1.Single_set={(k,15,17)}
  4. From Eq.[1],gen(18)= ϕ $\rightarrow$ No new SLVs
- Slicing_Function fetches $(k, 15, 17)$
  1. period((k,15,17),obj1)=[15,14,13,12,11,20,19,18,17]
  2. DataDependentOn((k,15,17),obj1)=20 $\rightarrow$ Sliced_Lines={19,16,18,20}
  3. (k,15,17) is killed $\rightarrow$ obj1.Single_Set=ϕ
  4. From Eq.[1],gen(20)={m}
  5. From Eq.[9,10]: Start_Gen_Circular(20,obj1)=19, Stop_Gen_Circular(20,obj1)=11
  6. obj1.Single_Set={(m,19,11)}
- Slicing_Function fetches $(m, 19, 11)$
  1. period((m,19,11),obj1)=[19,18,17,16,15,14,13,12,11]
  2. DataDependentOn((m,19,11),obj1)=13 $\rightarrow$ Sliced_Lines={19,16,18,20,13}
  3. (m,19,11) is killed $\rightarrow$ Single_Set = ϕ
  4. gen(13) = ϕ
- Because there is no longer SLVs in $obj_1.Single_Set$.So, Slicing_Function stops

Example 10 (Fig.8(B))**.** obj1 is a circular object. if we assume that we have the slicing criterion: $< a, 53 >$. Then the Slicing Function will do the following steps:

- obj3.Statement_Set_IF=[41,42,43,44], obj3.Statement_Set_ELSE=[47,48,49,50,51,52,53,54]
- From slicing criterion $< a, 53 >$: obj3.Single_Set={(a,53,47)} .Eq.[6,7] are used to calculate *start* and *stop* properties.
- obj3.Slicing_Function has fetched (a,53,47):
  1. period((a,53,47),obj3)={53,52,51,50,49,48,47}
  2. DataDependentOn((a,53,47),obj3)=53 $\rightarrow$ Sliced_Lines={53}
  3. (a,53,47) is killed $\rightarrow$ Single_Set = ϕ
  4. gen(53)={b,c} $\rightarrow$ obj3.Single_Set=(b,52,47),(c,52,47).
- Slicing_Function fetches (b,52,47)
  1. period((b,52,47),obj3)={52,51,50,49,48,47}.
  2. DataDependentOn((b,52,54),obj3)=51 $\rightarrow$ Sliced_Lines={53,51}
  3. (b,52,47) is killed $\rightarrow$ obj3.Single_Set={(c,52,47)}.
  4. gen(51)={d} $\rightarrow$ obj3.Single_Set= {(c,52,47),(d,50,47)}
- Slicing_Function fetches (c,52,47)
  1. DataDependentOn((c,52,54),obj3)=47 $\rightarrow$ Sliced_Lines={47,53,51}
  2. (c,52,47) is killed $\rightarrow$ obj3.Single_Set={(d,50,47)}

3. gen(47)= ϕ

- Slicing_Function fetches (d,50,47)
  1. DataDependentOn((d,50,47),obj3)= ϕ
  2. (d,50,47) is killed $\rightarrow$ obj3.Single_Set= ϕ

The lines that are sliced from obj3 are:{47,51,53}

4.9 Detecting the Data Dependencies between the Objects

The interfaces are used to reproduce the SLVs in two cases. Let us setup the following common assumption in order to illustrate these two cases:

- there are two objects(*obj1* & *obj2*)
- In the program, there are two statements: ($stm1 \in obj1$ and $stm2 \in obj2$)
- there is an interface: interface(*stm2*,*stm1*). Notice here that stm2 is the FROM side.
- The slicing function of obj2 has fetched slv_i from obj2.Single_Set
- $stm2 \in period(slv_i)$.
- if the slicing function visits stm2 in order to check the condition: ($kill(stm2) = slv.var$). At this point, we have two cases:
  1. **if stm2 does not satisfy the condition** ($kill(stm2) = slv.var$)**:** then $slv_i.var$ is reproduced in obj1. Take in your consideration that the reproducing is an *insertion* case when *start* and *stop* properties are calculated for slv_i in obj_1.
  2. **if stm2 satisfies the condition** ($kill(stm2) = slv.var$)**:**, then the Slicing Function of obj2 kills slv_i by removing it from obj2.Single_Set. Then the Slicing Function of obj2 applies the function: $gen(stm2)$ in Eq.1. All the new SLVs, which are generated from stm2, are reproduced in obj1.

```
1.  void func1() {
2.       int y=0;
3.       int k=9;
4.       int m=k-3;
5.       int x=10;
6.       if(m<4) {
7.                y=40;
8.                while(x<1000) {.
9.                          x=x+20;
10.                         y=50;
11.                                 }
12.               r=50;
13.               j=y+3;
14.               }
15. }
```

(A)

```
1.  void func1() {
2.       int y=0;
3.       int k=9;
4.       int m=k-3;
5.       int x=10;
6.       if(m<4) {
7.                y=40;
8.                while(x<1000) {.
9.                          x=x+20;
10.                         y=50;
11.                                 }
12.               r=y+4;
13.               j=r+3;
14.               }
15. }
```

(B)

Figure 9:

In both of the two source codes in Fig.9(A), we have two main objects, obj1.Statement_Set=[6,7,12,13]. obj2.Statement_Set=[8,9,10]. In addition, there are two interfaces. The first interface is from obj1 to obj2: (12,10) and the second interface is from obj2 to obj1: (8,7)

Example 11 (Fig.9(A))**.** By assuming that the slicing criterion is $< j, 13 >$

- obj1.Single_Set={(j,13,6)}
- period((j,13,6),obj1)=[13,12,7,6]

- Because DataDependentOn((j,13,6),obj1)= 13 $\xrightarrow{then}$(j,13,6) is killed from obj1 and (y,12,6) is generated
- Single_Set={(y,12,6)}.
- period((y,12,6),obj1)=[12,7,6]
- Slicing_Function of obj1 visits orderly the statement 12, 7, then 6 to find the first statement statisfying the the condition $kill(period[m]) = y$
- The Slicing_Function of obj1 visits statement 12, because stmt12 does not satisfy the condition $kill(12) = y$ and because statement 12 is the FROM side in the interface (12,10) $\xrightarrow{then}$ 'y' is reproduced in obj2.
- obj2.Single_Set += {(y,10,8)}. *start* and *stop* properties are calculated from Eq.[6,7]
- The Slicing_Function of obj1 goes on visiting the rest of the period ..
- The Slicing_Function of obj2 is fired later..

Example 12 (Fig.9(B))**.** By assuming that the slicing criterion is $< j, 13 >$:

- obj1.Single_Set={(j,13,6)}
- Slicing_Function of obj1 fetches (j,13,6)
  - period((j,13,6),obj1)={13,12,7,6}
  - DataDependentOn((j,13,6),obj1)= 13 $\xrightarrow{then}$ statement 13 is sliced.
  - (j,13,6) is killed and (r,12,6) is generated
  - obj1.Single_Set={(r,12,6)}.
- Slicing_Function of obj1 fetches (r,12,6):
  - period((r,12,6),obj1)={12,7,6}
  - DataDependentOn((r,12,6), obj1)= 12 $\rightarrow$ statement 12 is sliced.
  - (r,12,6) is killed in obj1 and (y,11,13) is created $\rightarrow$ obj1.Single_Set={(y,11,6)}
  - y is reproduced in obj2 $\rightarrow$ obj2.Single_Set={(y,10,8)}
- Slicing Function of obj1 fetches (y,11,6). period((y,11,6),obj1)=[7,6] ....*continue as before*
- Slicing Function of obj2 fetches (y,10,8). period((y,10,8),obj2)=[10,9,8] ....*continue as before*

4.10 Detecting the Control Dependencies

In order to find all the statements that the slicing criterion is directly or indirectly a control dependent on :

1. One of the statements in the object is sliced
2. The Slicing_Function tests whether the header line is sliced before. if not:
   (a) The header line is sliced.
   (b) The statements that determine the scope of the corresponding block like BEGIN ... END or the curly braces '{' '}' are also sliced.
   (c) All the variables involved in the header line are generated as SLVs and are added to the Single_Set of the object.
   (d) The object sends a signal to the parent object in order to let it repeat the steps from (2).

Example 13 (Fig.10)**.** There are three objects: MainTrack, obj1, obj2. MainTrack.Statement_Set=[1,2,3,4,5,6]. obj2.Statement_Set=[7,8,9]. obj3.Statement_Set=[10,11,12,13]

if we assume that there is the unique slicing criterion = $< k, 13 >$, then:

- The SLV 'k' is inserted at statement 13. Thus, obj2.Single_Set={(k,13,10}.
- From Eq.15, period((k,13,10),obj2)=[13,12,11,10]
- DataDependentOn((k,13,10),obj2)=13:

```
1.  void main() {
2.        int y=0;
3.        int k=9;
4.        int m=k-3;
5.        int x=10;
6.        m=20;
7.           if(m<4) {
8.                 m = 25;
9.                 y   = 20;
10.                while(x<1000) {
11.                         y = y + 1;
12.                         x=x+20;
13.                         k=y+40;
14.                                  }
15.                        }
16.              }
17. }
```

Figure 10:

1. Statement 13 is sliced
2. (k,13,10) is killed and (y,12,10) is generated
3. obj2.Single_Set={(y,12,10)}
4. The statements 10, 14 are sliced
5. gen(10) $= x \xrightarrow{then}$ obj2.Single_Set={(y,12,10),(x,13,11)}
6. Due to the existence of the interface (10,9) and to generating (x,13,11) $\xrightarrow{then}$ obj1.Single_Set={(x,9,7)}
7. Because obj2.ID_Parent = obj1 $\rightarrow$ obj2 sends a signal 'Slice_Header' to obj1
8. obj1 checks whether its Header statement is sliced, if it is not sliced, then:
   - statement 7 and 15 are sliced.
   - gen(7) $= m \xrightarrow{thus}$ obj1.Single_Set={(x,9,7),(m,7,7)}
   - because obj1 is connected with MainTrack by the interface(7,6) and statement 7 has generated the new SLV(m)$\xrightarrow{then}$ the SLV m is reproduced in MainTrack object
   - MainTrack.Single_Set={(m,6,1)}
   - object1.Parent_ID= MainTrack. The header line of MainTrack is sliced by default.

4.11 IF .. ELSE block

Inserting a child block like IF block or WHILE block in a parent block does not damage the original data dependencies in the parent block. This is because the slicing is a MAY solution which means all of the statements that may influence the slicing criterion are sliced. On other side, IF block and WHILE blocks have an only one branch which it might be executed or not. Hence, the original data dependencies in the parent block is not affected by inserting a child block except for some cases.

Example 14 (Fig.11)**.** Notice that in Fig.11(A): the statement 8 is a data dependent on line 7. However, in Fig.11(B), a WHILE child block is inserted between the lines (7,8) in Fig.11(A). The outcome of this insertion is shown in Fig.11(B). Notice that in Fig.11(B), there is also a data dependency from line 12(8 previously) to 7. Hence, inserting a child WHILE block does not damage the original data dependencies. For this reason, in calculating the period and concluding the Statement_Set sequenced sets, statement 12 (as an example) is considered an immediate successor to statement 7.

Inserting an IF..ELSE block is an exception to the above assumption in some cases. The problem of IF..ELSE block is that it has two branches and one of them must be executed. Hence, if there is a variable (x) defined in the two branches of a child IF_ELSE block, then the SLV (x) in the parent block can not skip the child IF_ELSE block like what (y) does in Fig.11(B). Assume that we have the following assumption:

1. There is a parent object called ob_{parent} and it has two consecutive statements: ℓ_1 and ℓ_2
2. An IF_ELSE object: ob_{child} is inserted in ob_{parent} between ℓ_1 and ℓ_2
3. The variable x is defined in both of the two branches of the child IF_ELSE block.
4. if x variable is generated as an SLV in ob_{parent} and its period has the two statements: ℓ_1 and ℓ_2. Then, the period of x must be truncated after ℓ_2

   Defined_Two_Branches property This set exists only in IF_ELSE object. This set is filled by all the variables that are defined in both of the two branches in the corresponding IF_ELSE block. variable y in Fig.12 is an example.

Eq.15 can be improved in order to handle this issue:

$$\begin{aligned} period(slv, obj_k) = & period_initial(slv, obj_k)[i] \mid period_initial(slv, obj_k)[i] \in obj_k \wedge \\ & (\neg \exists\, i > m \mid period_initial(slv, obj_k)[m] \in obj_{if_else} \wedge \\ & slv.var \in obj_{if_else}.Defined_Two_Branches) \\ & where\, i\, and\, m\, are\, indexes\, in\, period_initial(slv, obj_k) \end{aligned} \quad (17)$$

According to Eq.17, all the statements in the period, which come after a child IF_ELSE block defining in its two branches the current SLV, are removed from the period. The coming example illustrates this concept:

Example 15. In Fig.12, statement 16 is not a data dependent on line 6 because y is defined in both of the two branches of the IF..ELSE block and one of these two branches is indeed executed.

Example 16 (Fig.12)**.** There are two objects:

1. obj1 is the parent block and its Statement_Set=[5,6,15,16]
2. obj2 is an IF_ElSE block and its Statement_Set=[7,8,9,12,13]

period_initial((y,15,5),obj1)=[15,13,12,11,9,8,7]. Statement 13 has index 2 in period_initial set. Because statement 13 belongs to an IF_ELSE block that defines the target SLV 'y' in both of its two branches, thus, period((y,15,5),obj1) does not accept from period_initial((y,15,5),obj1) any statement that its *index* ≥ 2. Accordingly, period((y,15,5),obj1)=[15]

```
1.  void func1() {
2.        int y=0;
3.        int k=9;
4.        int m=k-3;
5.        int x=10;
6.        if(m<4) {
7.              y=40;
8.              r=y+4;
9.              j=r+3;
10.             }
11. }
```

(A)

```
1.  void func1() {
2.        int y=0;
3.        int k=9;
4.        int m=k-3;
5.        int x=10;
6.        if(m<4) {
7.              y=40;
8.              while(x<1000) {
9.                    x=x+20;
10.                   y=50;
11.                         }
12.              r=y+4;
13.              j=r+3;
14.             }
15. }
```

(B)

Figure 11:

```
4.  ...
5.        if(m<4) {
6.              y=40;
7.              if(x<1000) {.
8.                    x=x+20;
9.                    y=50;
10.                      }
11.             else {
12.                   x=30;
13.                   y=20;
14.                 }.
15.              h=20;
16.              j=y+3;
17.             }
18. }
```

Figure 12:

5 The Algorithm

The *Slicing Program* is the main algorithm that is in charge of slicing all the lines in the source code. The *Slicing Function* is in charge of detecting the data dependencies in its object.

5.1 The Scanning Stage

The scanning stage builds and fills the intermediate data structure -*Objects* and *Interfaces*- where the slicing algorithms apply its analysis on.

5.2 The Main Algorithm - Slicing Program

The Algorithm of the Slicing Program is illustrated in Fig.13. The *OBJECTS* input refers to the objects that are constructed in the scanning stage. *SLICING_CRITERIA* input is a collection contains the slicing criteria, which each is a pair of $< var, location >$. The *SLICED_STATEMENTS* output are the code statements that are sliced by the Slicing Functions.
Slicing Program begins the analysis in considering the variable in each slicing criterion as SLV. The function FindObject returns the object that the *sc.location* of the slicing criterion is one of its statements. The function Insert_SLV injects the *var* of the slicing criterion in the single_set of the found object. Insert_SLV uses the equations [6,8,7] in calculating *start* and *stop* properties.
Afterward, the Slicing_Function of each object is invoked. Since the termination condition is the emptiness of all the *single_set*s, then, the function IsEmpty checks this condition over all the objects. Line 11 setup a new iteration if any *ob.single_set* is not empty.
It is worth noting that Each Slicing Function is responsible in killing its SLVs, but sometimes, the object receives new SLVs from other objects. Thus, the Slicing Program may make many iterations until satisfying the termination condition.

SlicingProgram(*OBJECTS*, *SLICING_CRITERA*)
Input *OBJECTS* -
  SLICING_CRITERA -
Output *SLICED* - the modified code
1. **foreach** *sc* **in** $SLICING_CRITERA$ **do**
2.   *object* = FindObject(*sc.location*)
3.   Insert_SLV(*object*, *sc.var*, *sc.location*)
4. $flag_new_Iteration := true$
5. **while** $flag_new_Iteration$ **do**
6.   **foreach** $ob \in OBJECTS$ **do**
7. $SLICED :=$ Slicing_Function($ob, SLICED$)
8. $flag_new_Iteration = false$
9.   **foreach** $ob \in OBJECTS$ **do**
10.   **if** !IsEmpty(*ob.single_set*)**then**
11. $flag_new_Iteration := true$
12. return SLICED

Figure 13: The Algorithm of SlicingProgram

5.3 The Algorithm of the Slicing Function

Fig.14 shows the algorithm of the SLICING_FUNCTION. SLICING_FUNCTION fetches the SLVs from the *single_set* individually,(line 1) . The current SLV is stored in *slv*. The *period* path of *slv* is calculated by PERIOD function (line 2) . The inputs of PERIOD function are the SLV in its triple format as well as the object that the SLV is stored in it.
Afterward, the SLICING_FUNCTION starts orderly visiting the statements in the *period*, the current visited statement is (ℓ)(line 3). In (line 4) , (ℓ) is checked whether it is a FROM side of one of the interfaces, if yes, the interface is stored in the variable $\imath$.

line 5 checks whether the current visited statement ℓ kills *slv*, if no(line 5) , then the SLICING_FUNCTION reproduces *slv* in the another side of the interface $\imath$, (line 6 and 7).
if ℓ kills *slv* (line 8) and if ℓ is not sliced before (*line 9,10*), then the following steps take place:

1. ℓ is sliced by adding it to *SLICED_STATEMENTS*, (line 11)
2. New SLVs are generated from ℓ, and all of these SLVs are added to *ob.Single_Set* after calculating their *start* and *stop* properties(lines 12,13,14,15,16).
3. if the HeaderLine of the object is not sliced before, then it is sliced as well as the determiners of its scope. Then:
   - All of the variables involved in the *ob.HeaderLine* are generated as SLVs and added to the *ob.single_Set* (line 17,18,19,21,22,23,24)
   - If the *ob.HeaderLine* is a FROM side in an interface connecting obj2 with ℓ, then all the new SLVs are reproduced in obj2 (line 19,20,21,22,23,24,25,26).
4. the object sends a signal to its parent object in order to slice its *ob.HeaderLine* if it is not sliced before. (lines 27,28)

slv is killed by removing it from the *ob.single_set*, (line 29). Finally, the SLICING_FUNCTION returns *SLICED_STATEMENTS* (line 30).

SLICING_FUNCITON($ob, SLICED$)

Input ob - The object which the Slicing Function is going to detect its internal dependencies
$SLICED_STATEMENTS$ - The set of the already sliced statements.

Output $SLICED_STATEMENTS$ -

1. **foreach** slv **in** $ob.single_set$ **do**
2. $period = $ PERIOD(slv, ob)
3. **foreach** $\ell \in period$ **do**
4. $\imath = $ INTERFACE_FROM(ℓ)
5. **if** $kill(\ell) \neq slv$ **then**
6. **if** $\imath \neq NULL$ **then**
7. REPRODUCE($slv.var, \imath$)
8. **else**
9. **if** $\ell \in SLICED_STATEMENTS$ **then**
10. **break**
11. $SLICED_STATEMENTS := SLICED_STATEMENTS + \ell$
12. $GEN_VARS := GEN(\ell)$
13. **foreach** $var \in GEN_VARS$ **do**
14. $start := $ START($\ell, ob, case.generation$)
15. $stop := $ STOP($\ell, ob, case.generation$)
16. $ob.single_set + = \{var, start, stop\}$
17. **if** $ob.HeaderLine \notin SLICED_STATEMENTS$ **then**
18. $SLICED_STATEMENTS + = ob.HeaderLine + Begin_Scope + End_Scope$
19. $GEN_VARS = $ GEN($ob.HeaderLine$)
20. $\imath = $ INTERFACE_FROM($ob.HeaderLine$)
21. **foreach** $var \in GEN_VARS$ **do**
22. $start := $ START($\ell, ob, case.generation$)
23. $stop := $ STOP($\ell, ob, case.generation$)
24. $ob.single_set + = \{var, start, stop\}$
25. **if** $\imath \neq NULL$ **then**
26. REPRODUCE($var, \imath$)
27. **if** $ob.PARENT \neq NULL$ **then**
28. SLICE_HEADER($ob.PARENT$)
29. REMOVE_SLV(slv, ob)
30. **return** $SLICED_STATEMENTS$

Figure 14: The Algorithm of SLICINGFUNCTION

6 Experiments

According to our estimations; the processing time of the slicing analysis has four main factors regardless of the applied approach; The number of the variables, the number of the lines, the number of the Fixed Point Iterations and the number of the blocks. Our experiments endeavor to study carefully the trace of each factor in both of the two slicing appraoches, *Objects-Based Slicing and PDG-Based Slicing.* A particular Program Generator Tool generates 21 source code files to be tested and measured.

6.1 Number of the Variables and Number of the Iterations

The first twelve experiments measure the effect of changing the number of the variables or/and the number of the fixed point iterations. As known, in all of the dataflow analysis techniques, a one fixed point iteration is not enough if there is a WHILE loop includes all the source code lines because all the reaching definitions records exist in the last exit point of the WHILE loop block must be copied again to the first entry point of the WHILE loop. if there is a nested two WHILE loops, then we need three fixed point iterations. The number of the lines of those twelve experiments is 1000 lines. The number of the variables vary from 10, 20, 40, to 100 and the number of the iterations vary from 1,2 to 3 iterations. Hence, the results of the measurements consists of an array where the rows have constant number of variables and the columns have constant number of iterations.
Table 1 shows the measurements of PDG-Based slicing. Table 2 shows the corresponding measurements in Objects-Based Slicing approach. Table 3 shows the amount of the speed-up that we get from applying objects-based slicing instead of PDG-Based slicing.

6.2 Excluding the Parsing Time

The definition of the parsing time is: "the required time to read and realize the type, the assigned variable and reference variables of each line. This process may be implemented by many ways. Our implementation aims to decrease the much of time consumed in comparing the strings by converting the strings of the variables to integers. The parsing time is identical for the two compared approaches -PDG-Based Slicing and Objects-Based Slicing-. The parsing time for the above 12 source code files appear in Table

From this table, we can calculate the analysis time in Table 1 and in Table 2 by subtracting from each cell the value in the corresponding cell in Table 4. However, from the new analysis values which do not contain the parsing times, the speed up values of the analysis times can be calculated and shown in Table

6.3 Variation in the number of the lines

Second group of files were designed to measure the effect of increasing the number of the source code lines. Table. 6 shows the measurments of different line number that are 100, 250, 500, 750 and 1000. The first and second columns shows the much of the time of 2 iterations, 20 variables and the line number which is written at the corresponding cell in the first column. The second and third columns compares the total analysis time for each file and the fourth column shows the SpeedUp for every file.

Table 1: PDG-Based Slicing - Variations in the Number of Variables and Iterations - 5000 Executions, 1000 Lines

	Number of Iterations		
Variables	1	2	3
10 vars	77,1734 sec.	125,3591 sec.	178,8082 sec.
20 vars	82,3397 sec.	132,5435 sec.	196,2672 sec.
40 vars	99,5716 sec.	171,1097 sec.	247,7401 sec.
100 vars	157,4370 sec.	362,0000 sec.	588,6480 sec.

Table 2: Objects-Based Slicing - Variations in the Number of Variables and Iterations - 5000 Executions, 1000 Lines

Variables	Number of Iterations		
	1	2	3
10 vars	29,0496 sec.	30,0817 sec.	28,3076 sec.
20 vars	31,5558 sec.	33,1718 sec.	34,1609 sec.
40 vars	39,7702 sec.	41,8113 sec.	43,9855 sec.
100 vars	59,7344 sec.	67,7538 sec.	69,0389 sec.

Table 3: Speed Up : Table1/Table2

Variables	Number of Iterations		
	1	2	3
10 vars	2,66	4,17	6,32
20 vars	2,61	4,00	5,75
40 vars	2,50	4,09	5,63
100 vars	2,64	5,34	8,53

Table 4: Parsing Times - Variations in the Number of Variables and Iterations

Variables	Number of Iterations		
	1	2	3
10 vars	24,2655 sec.	25,0994 sec.	24,9979 sec.
20 vars	27,7292 sec.	28,696 sec.	28,9642 sec.
40 vars	34,336 sec.	34,3204 sec.	35,1163 sec.
100 vars	48,077 sec.	48,525 sec.	48,534 sec.

Table 5: Speed Up Table - Analysis Time - 5000 executions and 1000 lines

Variables	Number of Iterations		
	1	2	3
10 vars	11,06	20,12	46,47
20 vars	14,27	23,20	32,19
40 vars	12,00	18,26	23,97
100 vars	9,38	16,30	26,34

Table 6: The Variation of the Lines Numbers

Lines	Total Time		
	PDG	Objects	SpeedUp
100	4,70 sec.	3,16 sec.	1,48
250	12,31 sec.	7,80 sec.	1,58
500	40,45 sec.	15,30 sec.	2,64
750	82,26 sec.	22,43 sec.	3,67
1000	132,54 sec.	33,17 sec.	4,00

6.4 Testing the Scalability

In order to test the scalability, the line numbers 984 and 985 are modified in the file that has 1000 lines, 100 variables and 3 iterations. Manually, these two lines are modified to:

```
a82=1;
a69=a82;
```

By conisdering the slicing criterion is $< a69, 985 >$, we get the following results: PDG-Based Slicing consumes 575.13 seconds/5000 executions . Analyzing this file by Objects-Based Slicing approach takes 49.275 seconds/5000 executions. The speed up of the total time is 11.67. This file is parsed by 48.68 seconds/5000 executions. if the speed up is calculated in terms of analysis time only, then we get a speed up equals 886.

6.5 Memory Saving

Table 7 shows the memory savings that is gotten by using Objects-Based Slicing rather than the conventional method: "PDG-Based Slicing".

The measurements exclude the *Parsing* time. in the PDG-Based slicing method, the data fields that are included in reserved-memory measurements are:

- The reaching definition records in every program point.
- The data dependencies edges. Every line points to the lines, which it is a data dependent on.
- The control dependency edge. Each line points to the line, which it is a control dependency on.
- The control flow edges of each program line.

The calculations and the measurements tend to favor PDG-Based Slicing method over Objects-Based Slicing. Thus, rather than considering every RD record consists of two fields; it is considered in our measurements as a one field only. This makes a significance drop in the Saving Rate.

The data fields in the Objects-Based Slicing included in the measurements are:

- The main properties of each block: *headerLine*, *endLine*, *startLine*, *indexParent*,*indexBlock*.
- The maximum size that the single-set of each object reaches.
- The interface set, specialized in reporting the SLVs reproduced through the interface.
- Two data fields stored in every line; *indexObject*, pointing to the object that the line belongs to, and interface_edge, if *Interface_edge* ≥ 0, then this line is a *From_point* side in an interface and *Interface_edge* points to the *To_point* in the interface.

Table 7: Memory Saving

File Name	PDG	Objects	Saving Rate
20 vars,100 line,2 Itr.	5,708	230	25
20 vars,250 line,2 Itr.	15,607	542	29
20 vars,500 line,2 Itr.	29,517	1048	28
20 vars,750 line,2 Itr.	44,292	1560	28
10 vars,1000 line,1 Itr.	30,864	2,234	14
10 vars,1000 line,2 Itr.	32,359	2,545	13
10 vars,1000 line,3 Itr.	33,717	2,711	12
20 vars,1000 line,1 Itr.	50,522	2,137	24
20 vars,1000 line,2 Itr.	56,334	2,939	19
20 vars,1000 line,3 Itr.	61,519	3,407	18
40 vars,1000 line,1 Itr.	89,282	2,074	43
40 vars,1000 line,2 Itr.	104,809	3,067	34
40 vars,1000 line,3 Itr.	127,914	4,627	28
100 vars,1000 line,1 Itr.	200,501	2,041	98
100 vars,1000 line,2 Itr.	301,045	6,147	49
100 vars,1000 line,3 Itr.	332,425	7,543	44

7 Results and Discussions

Tables [1,2] shows the execution times of twelve source codes, involving in the experiments. Those tables motivates the fact that the total execution time for both of the two methods is proportional to the number of the iterations as well as the number of the variables. Furthermore, table 6 illustrates the proportional of the execution time with the number of the lines.

Tables [3,6] illustrates that the Speed Up is proportional mainly to the number of the iterations as well as the lines. Less importantly, to the number of the variables.

The Parsing technique used in implementing both of the two methods is same. it mainly depends on Regular Expression Library in Visual C#.NET 2012. Because the Parsing technique could be replaced by a lighter technique and in order to accurately compare between the two methods, Table 5 shows the speed up of the Analysis Times, where the parsing is not included. The speed-up in table 5 is significantly increased to reach to 46 as a maximum and to 9 as a minimum.

Section 6.4 shows a speed up reaches to 886 for the file: file_100vars_1000lines_3Iterations.txt. This file is Working with small sizes of problems, such as the one mentioned in Section, shows an extraordinary significance. The speed up in this case 886. This proves that the analyses in Objects-Based grow with the problem.

References

[1] C. Sandberg,A.Ermedahl,J. Gustafsson ,B. Lisper, Faster WCET Flow Analysis by Program Slicing

[2] Ottenstein, K. and Ottenstein, L. 1984. The program dependence graph in a software development environment. In Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. ACM Press, New York, NY, USA

[3] J. Ferrante, K. Ottenstein, J. Warren,The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems, 1987

[4] F. Nielson, H. Nielson, C. Hankin, Principles of Program Analysis, Springer

[5] M. Weiser, Program Slicing,Proceeding ICSE '81 Proceedings of the 5th international conference on Software engineering

[6] J. Sliva, A Vocabulary of Program Slicing-Based Techniques, ACM Computing Surveys (CSUR), June 2012

[7] A. Lucia, Program Slicing: Methods and Applications, Proceeding. First IEEE international workshop on Source Code Analysis and Manipulation. 2001

[8] M. Weiser, Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[9] M. Weiser,, Programmers use slices when debugging, Communications of the ACM, vol. 25, 1982, pp. 446-452.

[10] M. Weiser, Program Slicing, IEEE Transactions on Software Engineering, vol. SE-10, no. 4, 1984, pp. 352-357.

[11] S. Danicic, M. Harman, and Y. Sivagurunathan, A parallel algorithm for static program slicing,Information Processing Letters, vol. 56, no. 6, 1995, pp. 307-313.

[12] H. Agrawal, On slicing programs with jump statements, ACM Sigplan Notices, vol. 29,no. 6, 1994, pp.302-312.

[13] T. Ball and S. Horwitz, Slicing programs with arbitrary control-flow, Proceedings of 1st International Workshop on Automated and Algorithmic Debugging, Linkping, Sweden, Springer Verlag, New York, 1993, pp. 106-222.

[14] M. Harman and S. Danicic, A new algorithm for slicing unstructured programs, Journal of Software Maintenance: Research and Practice, vol. 10, no. 6, 1998, pp. 415-441.

[15] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A Brief Survey of Program Slicing. SIGSOFT Softw. Eng. Notes, 30(2):136, 2005.

[16] Johan Kraft, Enabling Timing Analysis of Complex Embedded Software Systems, Mlardalen University Press Doctoral Theses, 2010