# Overlapping Flows

Husni Khanfar

School of Innovation, Design, and Engineering, Mälardalen University,
SE-721 23 Västerås, Sweden
`Husni.Khanfar@mdu.se`

**Abstract.** The Set of Overlapping Flows (SOF) is a data structure that consists of program flows wherein each flow overlaps or is overlapped with at least one other flow in the set. This data structure enables us to build approaches to computing the control dependencies in unstructured programs on demand. Forming this data structure on demand is challenging because it requires checking the overlap between each flow in this set and each flow in the program under analysis. As a result, any static program analysis approach built on this data-structured foundation is expensive in terms of time. It needs better time complexity.
Our previous works presented an invented program representation for on-demand computations: the Predicted Code Block graph (PCB graph). This graph considers conditional statements as its main blocks. This work enhances the PCB graph to obtain any demanded SOF from it. The experimental evaluations show that computing the SOFs from the PCB graph is fast and scalable.

**Keywords:** Static Program Analysis, Control Dependence, Demand-Driven, Predicated Code Block (PCB) graph, Set of Overlapping Flows (SOF), Softwre Maintenance, Control Flow Graph.

## 1 Introduction

Static Program Analysis examines the source code without running it, and one of its main methods is Abstract Interpretation. This method abstracts some properties from the semantic of the code statements to represent the program's behavior. Each property has an abstract domain unrelated to the concrete values of the variables in the program. Abstract Interpretation works mainly with variables (data). Thus, it is also called Dataflow Analysis.

For example, the simple program in Fig 1-A shows that the variable `a` at the end of the program might be defined at labels 2 or 4. Further, it concludes that the variable `z` is not defined after executing Label 1. The property here is the scope of each definition (assignment), and the domain of this property is $\{$*Defined, Not Defined, Not Known*$\}$. This analysis is a dataflow analysis, and it is called *Reaching Definition.* It allows us to understand where to use the variables and the possible assignments of the variables at each program point.

There are five steps to establish a dataflow analysis. The first generates dataflow queries from the semantics of the source code. In Fig 1-A, the query $(b, 1)$ is generated from Label 1, while $(a, 2)$ is generated from Label 2. The second step propagates the queries in the forward or backward flows. Since the program flows are the means of the analysis besides the statements, dataflow analyses build their algorithms on the Control Flow Graph(CFG), which represents the

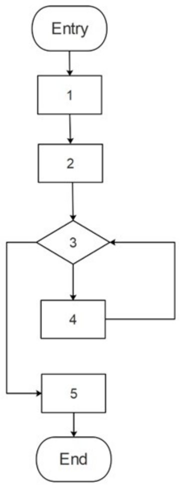
MRTC Report, Mälardalen Real-Time Research Centre - Mälardalen University - Sweden
ISRN: MDH-MRTC-349/2024-1-SE
Received in 1-Feb-2024 Accepted in 2-Feb-2024

statements as nodes and the flows as edges. In CFG, the nodes are connected by edges, and each node has two points, *entry point* and *exit point*. The third step tracks the propagation of the queries by saving a copy of each query at each program point it reaches. The fourth step applies a monotonic dataflow equation to each node. These equations constitute the entire mechanism that controls the propagation of the queries. Each equation associating with a node allows or prevents the queries from passing its node individually. Each equation has two inputs: the outputs from node neighbors and the node semantics. In the final step, dataflow equations run on iterations. In each iteration, the node equations run sequentially.

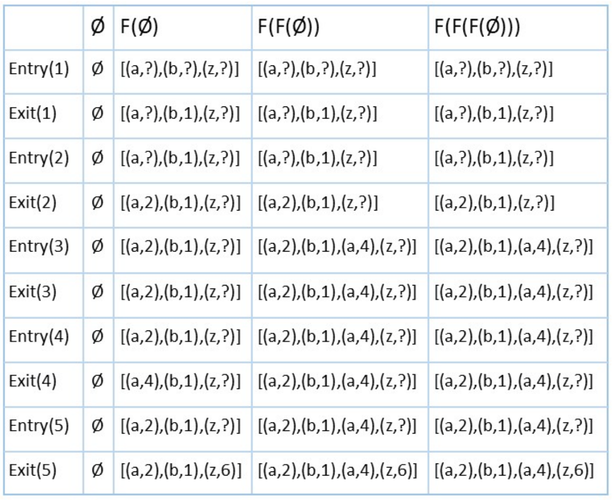
The question is whether one iteration is enough to propagate properly the dataflow queries. Suppose we have the following case: Node A's output is Node B's input, which is input to Node C, and Node C's output is Node A's input. In this case, when the equation of Node A runs in the first iteration, it does not have all the inputs, so its output is incorrect. Thus, we must make many iterations until all the equations become right. But what is the sign indicating that all equations are balanced between their inputs and outputs? This balance occurs when an iteration does not vary any node output. In this case, we reach the *Fixed-Point* and no more iterations are needed.

```
[b:=1]^1;
[a:=input()]^2;
while[a<10]^3
      [a:=a+1]^4;
[z:=a+b]^5;
END
```

(A) Source Code



(B) CFG



| | Ø | F(Ø) | F(F(Ø)) | F(F(F(Ø))) |
|---|---|---|---|---|
| Entry(1) | Ø | [(a,?),(b,?),(z,?)] | [(a,?),(b,?),(z,?)] | [(a,?),(b,?),(z,?)] |
| Exit(1) | Ø | [(a,?),(b,1),(z,?)] | [(a,?),(b,1),(z,?)] | [(a,?),(b,1),(z,?)] |
| Entry(2) | Ø | [(a,?),(b,1),(z,?)] | [(a,?),(b,1),(z,?)] | [(a,?),(b,1),(z,?)] |
| Exit(2) | Ø | [(a,2),(b,1),(z,?)] | [(a,2),(b,1),(z,?)] | [(a,2),(b,1),(z,?)] |
| Entry(3) | Ø | [(a,2),(b,1),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] |
| Exit(3) | Ø | [(a,2),(b,1),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] |
| Entry(4) | Ø | [(a,2),(b,1),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] |
| Exit(4) | Ø | [(a,4),(b,1),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] |
| Entry(5) | Ø | [(a,2),(b,1),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] | [(a,2),(b,1),(a,4),(z,?)] |
| Exit(5) | Ø | [(a,2),(b,1),(z,6)] | [(a,2),(b,1),(a,4),(z,6)] | [(a,2),(b,1),(a,4),(z,6)] |

(C) Iterative Reaching Definition Dataflow Analysis

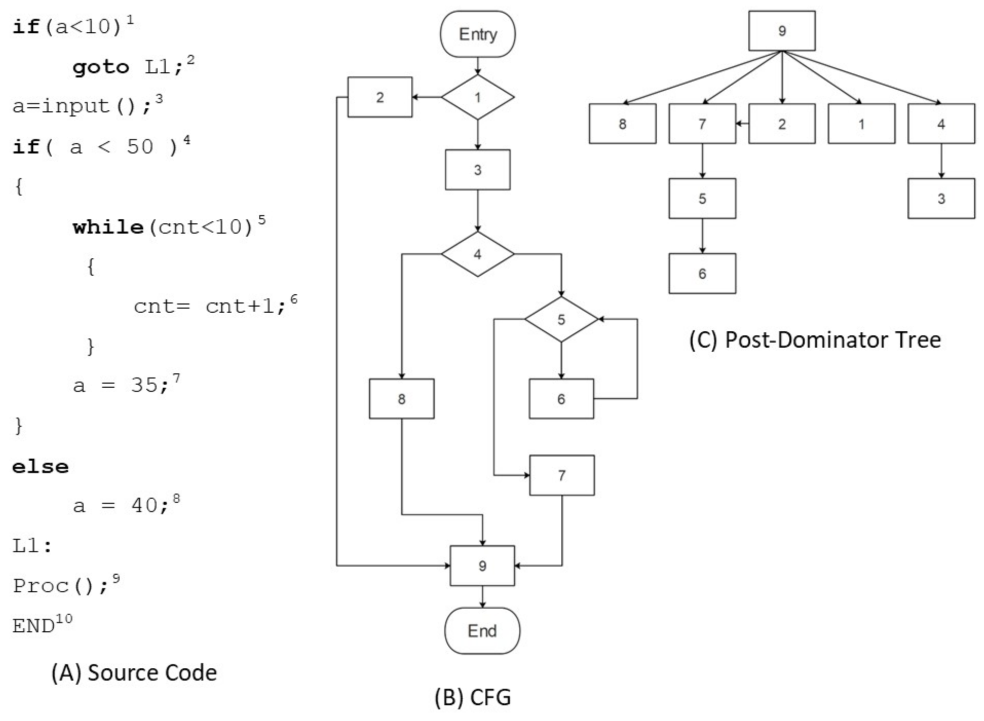
Fig. 1: Example of Dataflow Analysis - Reaching Definitions.

In Fig. 1, the first iteration $F(\phi)$ generates a dataflow query from each assignment in the format of (*var, label*) and then propagates it in all forward paths. The second iteration $F(F(\phi))$ produces different sets of queries in some points, such as Entry(3) and Exit(3), rather than $F(\phi)$. Thus, the dataflow analysis makes another iteration $F(F(F(\phi)))$. While the output of $F(F(F(\phi)))$ equals $F(F(\phi))$, thus causing reaching the Fixed-Point status, and there is no use in adding more iterations.

Control dependency is a relationship between a predicate and a statement, wherein the outcome of the predicate at the run-time determines the possible execution of the statement. The control dependency is calculated from the post-domination information. Node Z post-dominates Node X when all the paths from X to the END include Z. The predicate $p$ controls Node $\ell$ when

$\ell$ post-dominates one of the successors of $p$ and does not post-dominate the second. So, In Fig. 2, Label 4 controls Label 7 because Label 7 post-dominates Label 5 but does not post-dominate Label 8.

In order to obtain the predicates that control the execution of a particular Node in the CFG, all the post-domination information in the program should be obtained and arranged in a post-dominator tree. The control dependency is got from the tree by the following rule: Node W is control dependent on node U if the CFG contains an edge U $\rightarrow$ V, wherein W post-dominates V and W does not post-dominate U. For example, Fig. 2-C shows that Label 7 post-dominates Label 5, while Fig. 2-B shows the edge 4 $\rightarrow$ 5, and Fig. 2-C illustrates that Label 7 does not post-dominate Label 4. As a result, Label 4 controls Label 7.



```
if(a<10) 1
    goto L1; 2
a=input(); 3
if( a < 50 ) 4
{
    while(cnt<10) 5
     {
         cnt= cnt+1; 6
     }
    a = 35; 7
}
else
    a = 40; 8
L1:
Proc(); 9
END 10
```

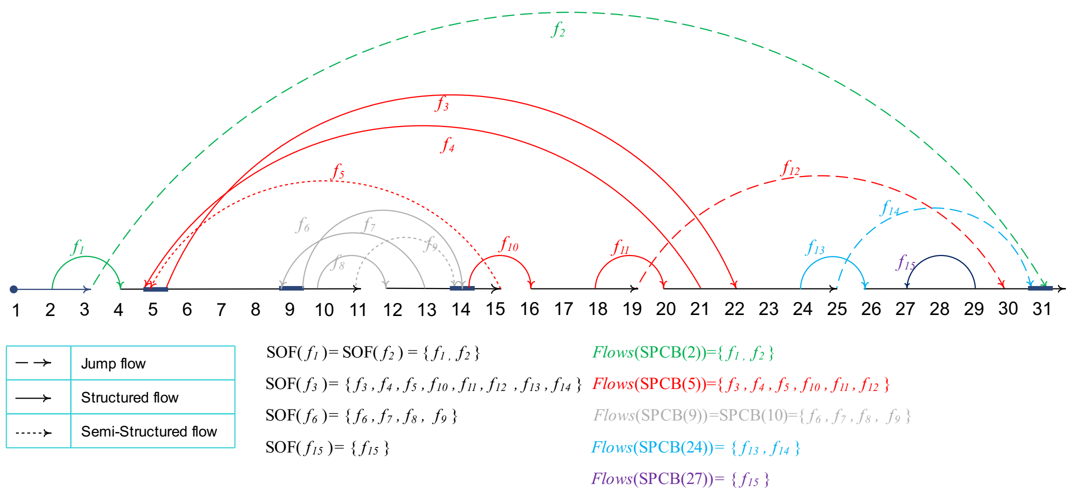
Fig. 2

Both dataflow analysis and control dependency computation based on a post-dominator tree are *comprehensive-based analysis solutions* under the following definition:

**Definition 1 (Comprehensive-Based Analysis Solution).** *is an analysis solution that could only find the correct property for one part if it finds the correct property for all other parts.*

Most static program analyses are comprehensive. In computing control dependence relations based on the post-dominator tree, all the post-domination facts in the entire CFG should be calculated first, even if the requested relations are for only one node. In the dataflow analysis, the computation of a node property might be affected if the output of another is not correct. Hence, making correct outputs for one node requires correct outputs for all. The problem with the comprehensive analyses is that they must do necessary and unnecessary computations. The contrary to this concept is the *demand-driven analysis*, which tries to confine its computations to the part under analysis.

Our previous works [1,2] eliminate unnecessary computations in the state-of-art approach. These works present an approach that can immediately capture the control dependencies in structured programs. Thanks to our newly proposed program representation, the Predicated Code Block graph (PCB-graph). The main feature of this graph is that it represents the conditional statements and does not discard them among different nodes, as CFG does. It preserves the syntactic structure (child-parent relationship) of conditional statements. Hence, there is no need to compute control dependence relations for structured codes. Afterward, our works [3,4] showed a new novel approach that computes control dependencies of unstructured programs. The new approach builds on a new type of information called the Set of Overlapping Flows, abbreviated by SOF and introduced in Definition 13.



| | |
|---|---|
| $\dashrightarrow$ | Jump flow |
| $\longrightarrow$ | Structured flow |
| $\dashrightarrow$ | Semi-Structured flow |

SOF($f_1$)= SOF($f_2$) = {$f_1$, $f_2$}
SOF($f_3$)= {$f_3$, $f_4$, $f_5$, $f_{10}$, $f_{11}$, $f_{12}$, $f_{13}$, $f_{14}$}
SOF($f_6$)= {$f_6$, $f_7$, $f_8$, $f_9$}
SOF($f_{15}$)= {$f_{15}$}

*Flows*(SPCB(2))={$f_1$, $f_2$}
*Flows*(SPCB(5))={$f_3$, $f_4$, $f_5$, $f_{10}$, $f_{11}$, $f_{12}$}
*Flows*(SPCB(9))=SPCB(10)={$f_6$, $f_7$, $f_8$, $f_9$}
*Flows*(SPCB(24))= {$f_{13}$, $f_{14}$}
*Flows*(SPCB(27))= {$f_{15}$}

Fig. 3: The label-axis graph of the source code in Fig.4

The program flow $\ell_a \to \ell_b$ overlaps $\ell_c \to \ell_d$ if and only if $\ell_a$ exists between $\ell_c$ and $\ell_d$, whereas $\ell_b$ does not fall between $\ell_c$ and $\ell_d$ (Section 3.1). For example, Fig. 3 abstracts the program flows of the source code in Fig. 4. It depicts them on the x-axis. It shows that $f_2$ overlaps $f_1$ and $f_5$ overlaps $f_{10}$. In addition, it shows that $f_1$ and $f_2$ constitute an SOF, while $f_6$, $f_7$, $f_8$, and $f_9$ constitute another SOF.

This paper introduces several vital contributions. The first is the Overlapping Theory, which shows that the predicates that control a particular statement exist only in one or a few SOFs. The second contribution is the Super Predicated Code Block (SPCB) unit. This unit is significant as it logically connects the PCBs (conditional statements), which plays a crucial role in our unique method of computing the Set of Overlapping Flows (SOFs) on the fly from the program representation PCB-Graph. Lastly, the paper includes experimental evaluations demonstrating the performance of computing the SOFs on the fly.

In addition to the contributions above, this work defines the Comprehensive Analysis. Furthermore, as a leading work in using SOFs and PCB graphs in static program analysis, this work provides a complete set of mathematical symbols and definitions expressing different parameters related to SOFs and PCB graphs. These symbols serve as concise and precise language to convey complex concepts and relationships in SOFs and PCB graphs.

In this work, the sections are organized as follows: Section 2 provides the background. Section 3 introduces the theory behind the Set of Overlapping Flows. Section 4 introduces the Super Predicated Code Block (SPCB) unit. Section 5 shows an algorithm for forming on-demand SOFs from the values of the edges in the CFG. Section 6 extracts the SOFs from the PCB graph with few computations. Section 7 provides experimental evaluations. Section 8 encompasses discussions regarding the performance of the algorithms. Section 9 discusses the related works. Finally, Section 10 sums up some conclusions.

## 2 Background

This section briefly describes the While language, the Control Flow Graph (CFG), which is the state-of-the-art program representation, the post-domination concept, and the control dependencies.

### 2.1 Small C Language Module

The Small C language is a subset of the C language used in this work to develop and test a new approach specialized in computing the control dependencies. This subset focuses on the parts in the C language that are related directly to the computation of the control dependencies, and it neglects all other parts, such as the pointers. The Small C language is a procedure $p$ having a statement $s$, which might be an elementary statement ($es$), conditional statement ($cs$), or a composite statement ($s_1; s_2$). Each elementary statement and a predicate of a conditional statement gets a unique integer label. The internal elementary statements in $s$ are labeled in ascending order according to their locations in the source code, from left to right and from top to bottom.

Let $n$, $c$, $f$, and $str$ denote integer, character, float, and string variables respectively. Let $proc$ denote a procedure name, $a$ denote an arithmetic expression, and the predicate $b$ denote a boolean expression. The abstract syntax of the Small C language is:

$$\begin{aligned}
v &::= n \;\mid\; c \;\mid\; f \;\mid\; str \;\mid\; b\\
type &::= \texttt{int} \;\mid\; \texttt{char} \;\mid\; \texttt{string} \;\mid\; \texttt{bool} \;\mid\; \texttt{void}\\
var\_list &::= v \;\mid\; var\_list', var\_list''\\
cs &::= \texttt{if (}\ [b]^{\ell}\ \texttt{)}\ \{s'\} \;\mid\; \texttt{if (}\ [b]^{\ell}\ \texttt{)}\ \{s'\}\ \texttt{else}\ \{s''\} \;\mid\; \texttt{while(}\ [b]^{\ell}\ \texttt{)}\ \{s'\} \;\mid\\
&\qquad \texttt{do}\ \{s'\}\ \texttt{while (}\ [b]^{\ell}\ \texttt{)} \;\mid\\
&\qquad \texttt{for (}\ [ds]^{\ell} \;\mid\; [a]^{\ell}\ ;\ [b]^{\ell'}\ ;\ [a]^{\ell''}\ \texttt{)}\ \{s\} \;\mid\\
&\qquad \texttt{switch (}\ [a]^{\ell}\ \texttt{)}\ \{\texttt{case}\ n' : s\ ; \texttt{break}; \texttt{case}\ n'' : s'; \texttt{break}; ...; \texttt{default} : s''\}\\
es &::= [x := a]^{\ell} \;\mid\; [\texttt{goto}\ \ell']^{\ell} \;\mid\; [\texttt{break}\ ]^{\ell} \;\mid\; [\texttt{continue}\ ]^{\ell} \;\mid\; [proc(var\_list)]^{\ell} \;\mid\; [type\ v]\\
s &::= es \;\mid\; s'; s'' \;\mid\; cs
\end{aligned}$$

This syntax abuses the notation and writes “predicate p” or “statement s” instead of “the label of predicate p” or “the label of statement s”.

```
//a,rslt,cntr,chr, LIMIT, and CNST are global vars.
void Proc(int*arr,int f1,int f2,int f2,int p1,int p2,int p3) {
    if( arr == null_ptr )
        goto L2;
    printf("**Calculations Started");
    while(true) {
        a=random(0,99);
        rslt=a^pw1+f1*a^pw2+f2*a^pw3;
        cntr=0;
        while(rslt>LIMIT)  {
            if ( ++cntr > 3 )
                break;
            rslt=rslt/2;
        }
        if( cntr>3 || arr[index]>0)
            continue;
        printf("\ncontinue? (Y|N)");
        chr = getchar();
        if( chr == 'N')
            goto L1;
        arr[a]=rslt;
    }
    printf("\nShow the elements of arr? (Y|N)");
    chrctr  = getchar();
    if( chrctr  == 'N')
        goto L2;
    cntr=0;
    do {
        printf("%d) %d \t",cntr,arr[cntr]);
    } while (++cntr1<100);
    L1: printf("\nNo more elements will be added to arr");
    L2: printf("\nProc End");
    }
```

Fig. 4: Running Example

### 2.2 The Flows of the Conditional Statements

There is a control flow edge from each elementary statement to its immediate next statement *next_stmnt*[1], and there are two control flow edges from each conditional statement to its immediate next statement. There is a set of program flows that form each type of conditional statement as follows:

- Suppose *cs* is an `if` that comprises a predicate $p$ and body $b$. Based on that, there are two control flows from $p$, wherein the first is from $p$ to the first statement in $b$, whereas the second is to *next_stmnt*. If the last statement in $b$ is not a `continue`, `break`, or `goto`, then there is a flow from the last statement in $b$ to *next_stmnt*.
- Suppose *cs* is an `if-else` conditional statement that comprises a predicate $p$ and two bodies $b_1$ and $b_2$. There are two flows from $p$; the first is to the first statement in $b_1$, and the second is to the first statement in $b_2$. In both of the two bodies, if the last statement is not `continue`, `break`, or `goto`, then there is a flow from the last statement of the body to *next_stmnt*.
- Suppose *cs* is a `while` or a `for` conditional statement that comprises a predicate $p$ and a body $b$. There are two flows from $p$, the first is to the first statement in $b$, and the second is to *next_stmnt*. In addition, there is a flow from the last statement in $b$ to $p$[2]
- Suppose *cs* is a `do .. while` conditional statement, then *cs* comprises a predicate $p$ and a body $b$. There are two flows from $p$; the first is to the first statement in $b$, while the second is to *next_stmnt*.
- The `switch` statement consists of many `case` predicates, each of which has a body end typically by the keyword `break`. There are two flows from each case; the first is to its immediate next `case` and the second to the first statement in its body.

### 2.3 The Control Flow Graph

The Control Flow Graph (CFG) for a program $s$ is a representation, using graph notation, to model the entire possible program flows in $s$. The CFG consists of nodes and edges; each node represents a predicate or an elementary statement, and each edge represents a possible program flow. The node is a label. The control flow edge in the CFG is formed by a pair of labels $(i, j)$, which means that $j$ might be executed immediately after $i$.

**Definition 2.** ***Control Flow Graph:*** *The Control Flow Graph for an intra-procedural program $s$ is a 4-tuple $(N, E, Entry, End)$.*

1. *$N$ is a set of nodes, where each node represents an elementary program statement in $s$.*
2. *$E$ is a set of program flows, where each program flow represents a possible program flow from one node to another. $E \subset (N \times N)$.*
3. *Entry: is a unique start node. $Entry \in N$.*
4. *End: is a unique exit node. $End \in N$.*
5. *There is a path from Entry to every $n \in N$.*
6. *There is a path from every $n \in N$ to End.*

[1] The immediate next statement is the first statement executed after the execution of a particular statement if no unstructured jump occurs.

[2] As a side note, this does not work if the last statement in the body is one of the jump statements (`goto`,`break`,`continue`), although this is a poor design.

### 2.4 Basic Definitions

In this subsection, we introduce few definitions that are related to the control dependencies.

**Definition 3.** ***Post-domination:*** *In a CFG G, any node n* post-dominates *node y if all the paths from y to Exit contain n.*

**Definition 4.** ***Standard Control Dependence:*** *In accordance to [5], node n is* standard control dependent *on node m in program s if:*

1. *There exists a non-trivial*[3] *path $\pi$ from $m$ to $n$ such that every node $n' \in (\pi - m, n)$ is post-dominated by $n$; and*
2. *$m$ is not strictly post-dominated by $n$.*

**Definition 5.** ***The Conditional Statement of a Label:***
*The conditional statement cs of a label $\ell$ refers to the innermost conditional statement where $\ell$ exists.*

### 2.5 Predicated Code Block Graph

Our previous works [1,2] introduced the notions of Predicated Code Blocks (PCBs) and PCB graphs. A PCB refers to the encapsulation of a predicate and the set of elementary statements in which the predicate controls its execution.

$$p ::= \{[b, es_1, \ldots, es_n], type\} \tag{1}$$

In addition to a predicate and a sequence of statements, PCBs carry types, *type*, signifying whether the PCB is linear, $L$, or cyclic, $C$. Intuitively, linear PCBs correspond to conditional statements, such as *if*, and cyclic PCBs correspond to iterative statements, such as *while*.

One of the most important features of PCB-graphs is that it converts the conditional statements to elementary statements by replacing their locations in their parent PCBs by a placeholder. All conditional statements are replaced by `skip` placeholders, except `if .. else` statements which are replaced by `in-child` placeholders.

The PCB is connected with its original location in its parent PCB by a uni-directional interface from its placeholder to the first statement in the PCB. In addition, the program flow that connects one of the statement in the conditional statement with the immediate next statement to this -as what is shown in Sec. 2.2- is converted to an interface.

The last point; the internal flow in the loop conditional statement which always occurs from the last statement to the first statement is not modeled because the data field *type*, whose value in this case is 'C' takes its role. Fig 5 is an example of a PCB-graph.

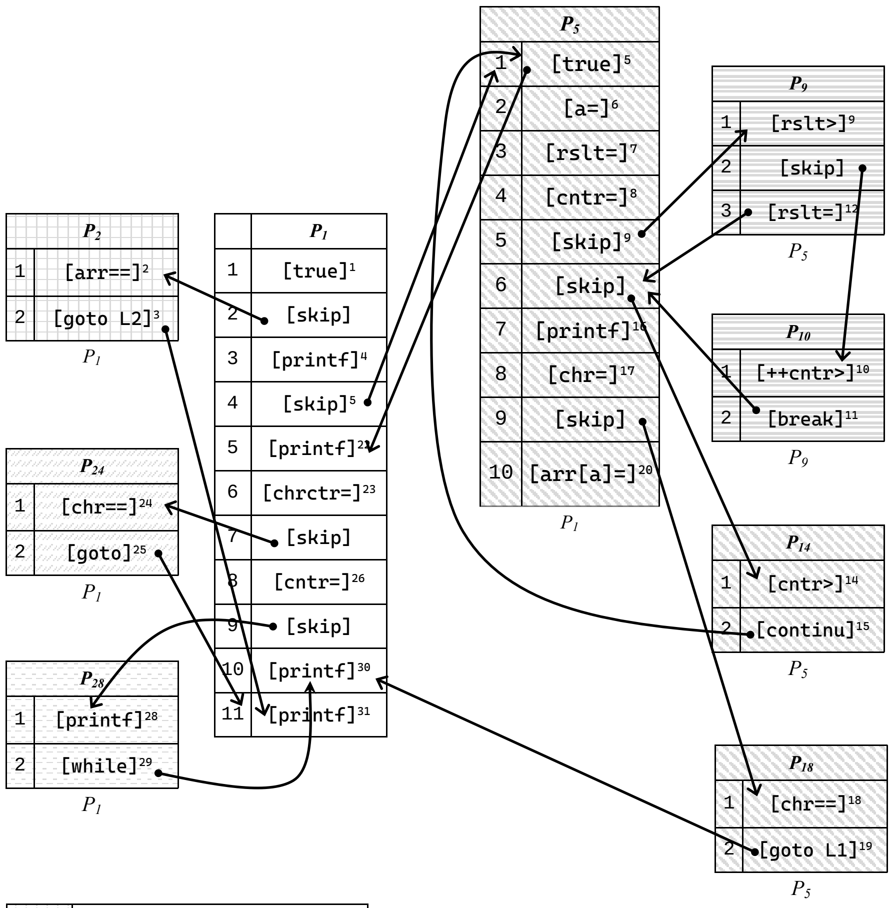
### 2.6 Fields in Labels, and PCBs

This paper use fields in each instance of Labels, and PCBs. These fields are:

- Labels:
  - Label_Instance.pcb: refers to the PCB representing the conditional statements that includes Label_Instance.
- PCBs:
  - PCB_Instance.parent: refers to the PCB representing the immediate outer conditional statement of the PCB_Instance conditional statement.
  - PCB_Instance.sof: refers to the SOF which includes the structured flows of PCB_Instance.

[3] Path $\pi$ is non-trivial if it contains at least two nodes [6]

[3] The differences between `skip` and `in-child` placeholders are related to the slicing approach shown in our previous works [1,2].



| | |
|---|---|
| ▦ | SPCB1: $P_2$ |
| ▨ | SPCB2: $P_5, P_{14}, P_{18}$ |
| ▤ | SPCB3: $P_9, P_{10}$ |
| ▤ | SPCB4: $P_{28}$ |
| ▨ | SPCB5: $P_{24}$ |

Fig. 5: The PCB graph of the source code in Fig.4

## 3 Overlapping Flows Theory

The beginning of this section specifies basic definitions related to overlapping flows. Then, it accurately determines the boundaries of the set of overlapping flow intervals, followed by a theorem that clarifies that the control dependency between a statement $\ell$ and predicate $p$ happens only when $p$ exists in a SOF bypassing $\ell$. The last theorem deals with many SOFs bypassing a statement $\ell$. It determines which of those certainly do not have predicates controlling $\ell$.

### 3.1 Basic Definitions

This paper shows how the interleaving between program flows could be a base for techniques in static program analysis. This section demonstrates standard definitions that mathematically formalize the fundamental concepts of overlapping flows.

**Definition 6.** ***The program flow notation*** $(\rightarrow)$ *refers to a pair of labels defining a program flow, such as:* $\ell \rightarrow \ell'$*, where* $\ell$ *is the outgoing label and* $\ell'$ *is the ingoing label.*

**Definition 7. Jump (Unstructured) Flow** *is a program flow whose outgoing label is a* jump statement *(e.g.* `goto`*).*

**Definition 8. Semi-Structured Flow** *is a program flow whose outgoing label is a* `break` *or* `continue` *statement.*

**Definition 9. Structured Flow** *is a program flow whose outgoing label is a predicate (e.g.* `while, for, if`*).*

The **label-axis** represents intra-procedural procedures on an X-axis. Rather than representing the statements as nodes, the label-axis represents them as ticks on an X-axis. It represents jump, structured, and semi-structured flows by unidirectional arcs from their outgoing labels (ticks) to their ongoing labels (ticks). The statements `goto`, `continue`, and `break` cut the X-axis because they do not have a flow to their immediate following statements on the program. Figure 3 shows a label-axis for an unstructured source code.

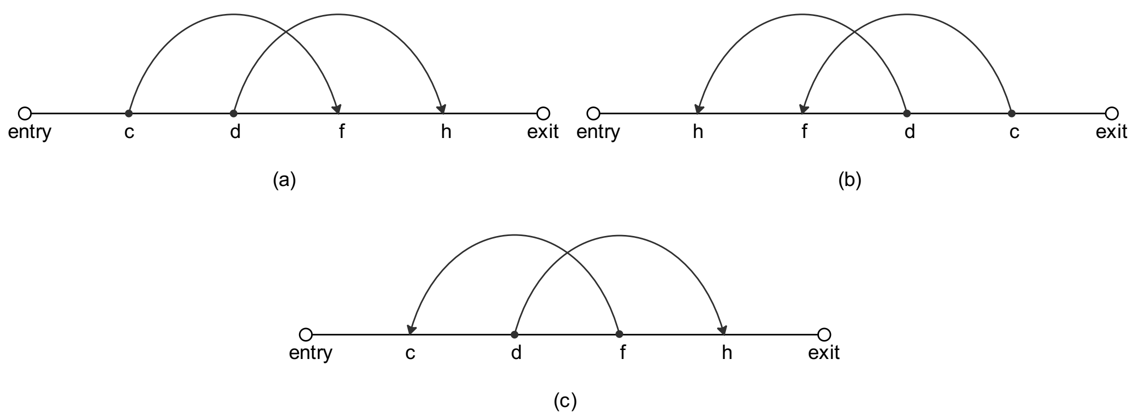


Fig. 6: Overlapping flows [3]

**Definition 10.** ***Bypassing:**[3] the program flow $j \rightarrow v$* bypasses *the label $t$ if either $j < t < v$ or $j > t \geq v$.*

**Definition 11.** ***Jump**($\ell$): is a function that returns the set of jump flows that bypass Label $\ell$.*

**Definition 12.** ***Overlapping Flows:**[3] the program flow $d \rightarrow h$* overlaps *$c \rightarrow f$ when $c > d \geq f$ or $c < d < f$ as well as $h$ is either less than $c$ and $f$ or larger than $c$ and $f$.*

Fig. 6 depicts the concept of overlapping.

**Definition 13. SOF** *is an abbreviation for a Set of Overlapping Flows. It consists of a finite number of program flows, wherein at least each flow overlaps with another flow.*

The flows in the SOF are collected inside two curly braces and separated by commas, for instance, $\{f_i, f_{i+1}, ..., f_n\}$.

**Definition 14.** ***SOF**$_i(\ell)$ is a SOF whose one of its flows bypasses the label $\ell$.*

The subscript here is added because many SOFs might bypass the statement under analysis $\ell$. So, the subscript here relates to the statement under analysis, not the SOFs.
In this context, *sof* denotes an SOF instance.

**Definition 15.** ***genSOF**($\ell$) is a function generating all the SOFs that bypass $\ell$.*

**Definition 16.** $\sum_{i=0}^{q}$***SOF**$_i(\ell)$ is the set of SOFs that each bypasses the statement $\ell$. The SOFs in $\sum_{i=0}^{q} SOF_i(\ell)$ are sorted according to the length of their intervals. Based on that, $SOF_0(\ell)$ has the shortest interval.*

The symbol SOF could be overloaded to express also the SOF that includes the structured flow of a PCB[4] as follows:

**Definition 17.** ***SOF**(**pcb**) the SOF that includes the structured flows of the PCB pcb.*

The expression $\sum$*SOF*($\ell$)-*SOF*($\ell$*.pcb*) means all the SOFs that bypass $\ell$ unless the one that includes the structured flows of the PCB $\ell$*.pcb*.

**Definition 18. Labels of SOF** *is the set of ingoing and outgoing labels of the flows belonging to a SOF.*

**Definition 19.** $\mathcal{L}$(***sof***) *is a function returns the Labels of sof sorted in ascending manner.*

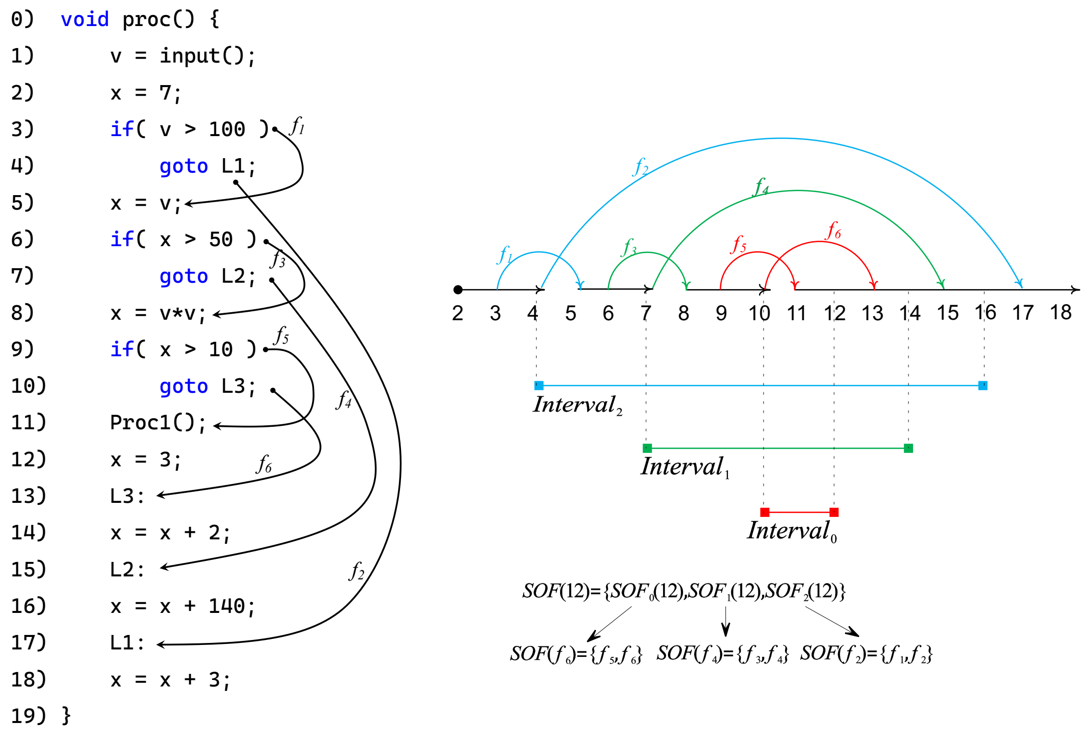
**Definition 20.** ***Len(sof)** is a function calculating the length of the SOF instance sof in accordance with the following equation: Max($\mathcal{L}$(sof))[5]-Min(sof)[6]*

**Definition 21.** $\mathcal{C}$(***sof***)[7] *is the set of the predicates whose structured flows in sof.*

The *SOF*($\ell$) without subscript means *genSOF*($\ell$) returns only single SOF equals $SOF_0(\ell)$.

**Definition 22.** ***Interval of a SOF** refers to the labels that are bypassed by the flows of a SOF.*

[4] Each PCB represents a conditional statement in the PCB graph. read Section 2.5
[5] Max is a function that returns the largest value in a numerical set.
[6] Min is a function that returns the smallest value in a numerical set.
[7] $\mathcal{C}$ is an abbreviation from *Condition*



```
0)  void proc() {
1)       v = input();
2)       x = 7;
3)       if( v > 100 )
4)            goto L1;
5)       x = v;
6)       if( x > 50 )
7)            goto L2;
8)       x = v*v;
9)       if( x > 10 )
10)           goto L3;
11)      Proc1();
12)      x = 3;
13)      L3:
14)      x = x + 2;
15)      L2:
16)      x = x + 140;
17)      L1:
18)      x = x + 3;
19) }
```

Fig. 7: Example - Lemma 8

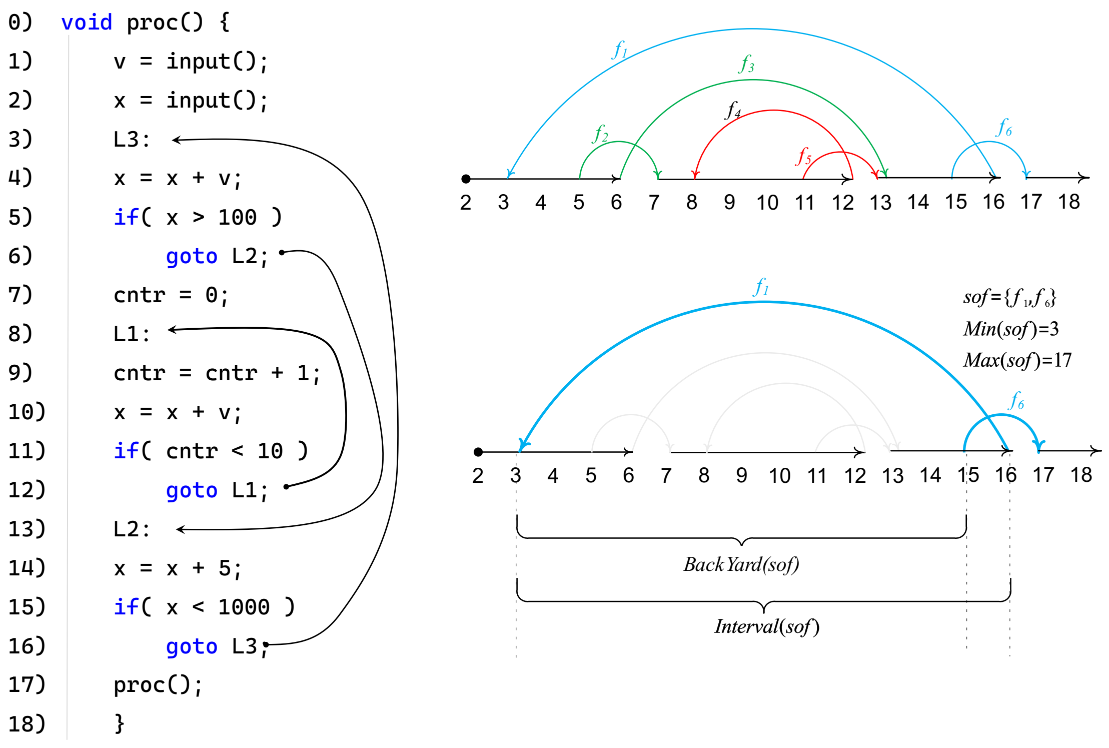
**Definition 23.** *$\mathcal{I}(\boldsymbol{sof})$ is a function that returns a set of the labels in the interval of the SOF instance sof. The labels in the set are sorted ascendingly.*

In Fig. 7, $genSOF(12)$=$\sum_{i=0}^{2} SOF_i(12) = \{SOF_0(12)$+$SOF_1(12)$+$SOF_2(12)\}$=$\{\{f_5, f_6\}$,$\{f_3, f_4\}$, $\{f_1, f_2\}\}$. The SOF instances are sorted according to the length of their intervals. The subscript $i$ in $SOF_i(\ell)$ refers to the order of the SOF among the others for a particular label. So, this subscript is relative. In Fig. 7, provided $sof$=$\{f_1, f_2\}$, then $sof$=$SOF_0(4)$=$SOF_1(7)$=$SOF_2(12)$. $\mathcal{L}(sof)$=$\{3,4,17\}$, $\mathcal{C}(sof)$=$\{3\}$, $Min(sof)$=$\{3\}$, $Max(sof)$= $\{17\}$. $Len(sof)$=14.

**Definition 24 (BackYard of SOF).** *refers to a subset of labels in the Interval of a SOF, which includes each label before the smallest outgoing label of a forward flow in the SOF.*

**Definition 25 ($\mathcal{B}(\boldsymbol{sof})$).** *This function returns the set of the BackYard of the SOF instance sof.*

In Fig. 8, provided $sof$=$\{f_1, f_6\}$, $\mathcal{B}(sof)$ = [3,15].
Finally, it is important to mention that if $\mathcal{I}(sof)$=[k,v[, and its statement is `while` or `for`, then $\mathcal{B}(sof)$=[k,k]. In other words, the predicates at the beginning of intervals are in the backyards because their forwarded flows run after them and not before, causing other predicates to reach and control them. For example, In Fig. 4, Label 5 controls the execution of Label 9. Thus, if Label 9 is at the beginning of its SOF, then it belongs to its backyard.



```
0)  void proc() {
1)      v = input();
2)      x = input();
3)      L3:
4)      x = x + v;
5)      if( x > 100 )
6)          goto L2;
7)      cntr = 0;
8)      L1:
9)      cntr = cntr + 1;
10)     x = x + v;
11)     if( cntr < 10 )
12)         goto L1;
13)     L2:
14)     x = x + 5;
15)     if( x < 1000 )
16)         goto L3;
17)     proc();
18)     }
```

Fig. 8: Example - Lemma 9

### 3.2 The Boundaries of Intervals

So, if we say that $\mathcal{I}(sof)$=[k,v[, that means the interval of *sof* is from $k$ to $v$. The two boundaries of the intervals might be included or excluded from the interval itself. As in mathematics, closed square brackets, "[..." or "...]", denote including, whereas open square brackets,"]..." or "...[", denote excluding. Selecting the closed or open bracket depends on satisfying Def. 22. Let us start with an example:

In Fig. 7, if we denote to the SOF $\{f_1, f_2\}$ *sof*, then $\mathcal{I}(sof)$=]3,17[. The left and right open squares indicate that neither $f_1$ nor $f_2$ bypass Label 3 or 17. In Fig. 8, If we denote to the SOF $\{f_1, f_6\}$ *sof´*, then $\mathcal{I}(sof')$=[3,17[. This interval starts with a closed square brace because $f_1$ bypasses Label 3, while $f_6$ does not bypass Label 17. If it is unimportant to know whether one of the boundaries is open or closed, we could use the notation |. However, the following theorem determines the boundaries of SOFs precisely.

**Lemma 1** *If sof is a SOF, then none of the program flows in sof bypasses Max($\mathcal{L}$(sof)).*

*Proof.* If *Max*($\mathcal{L}$(*sof*)) is bypassed by a flow in *sof*, this means it is not the maximum label in *sof*. This fact contradicts the assumption of the Lemma. □

**Lemma 2** *Suppose sof is a SOF, f ∈ sof, and Min($\mathcal{L}$(sof)) = f.out. No flow in sof bypasses Min($\mathcal{L}$(sof)).*

*Proof.* There are two cases: the first is $f.out < f.in$. In this case, if $f' \in sof$ and it bypasses $f.out$, then $Min(\mathcal{L}(sof)) \neq f.out$, and $Min(\mathcal{L}(sof))=Min(f.out,f.in)$. This case contradicts the assumption in the Lemma; therefore, it could not happen. The second case is when $f.out > f.in$. This case could not occur because $f.out = Min(\mathcal{L}(sof))$. Hence, the Lemma is right. □

**Lemma 3** *Suppose sof is a SOF, $f \in sof$, and $Min(sof) = f.in$. No flow in sof bypasses $Min(sof)$ except $f$.*

*Proof.* If the execution flow uses $f$, the program runs $f.out$ and then $f.in$. Based on that, $f$ bypasses $f.in$, and $f.in$, which equals $Min$(sof), is bypassed by one of the flows in *sof*. The proof of Lemma 2 can also prove that all other flows except $f$ can not bypass $Min$(sof). □

**Theorem 1.** *Suppose sof is a SOF, $f \in sof$, and $\mathcal{I}(sof)$=|k,v[, wherein $k$=$Min(\mathcal{L}(sof))$ and $v$=$Max(\mathcal{L}(sof))$. Based on that, if $k$==f.out, $k \notin\mathcal{I}(sof)$. If $k$==f.in, $k \in\mathcal{I}(sof)$, and always $v \notin\mathcal{I}(sof)$.*

*Proof.* Lemma 1 proves that $v$ always does not belong to $\mathcal{I}(sof)$. So, $\mathcal{I}$(sof)=|k,v[. Lemma 2 proves that if $k == f.out$, then $k \notin\mathcal{I}(sof)$, and $\mathcal{I}$(sof)=]k,v[. While Lemma 3 proves that if $k == f.in$, then $k \in\mathcal{I}(sof)$, and $\mathcal{I}(sof)$=[k,v[. □

## 3.3 The Locations of Predicates Controlling a Statement

Herein, we prove that all the predicates that control a statement under analysis belong only to an SOF bypassing this statement.

**Lemma 4** *Suppose sof is a SOF, $\ell \in$sof and $\mathcal{I}(sof)$=|k,v[. Then $v$ post-dominates $\ell$.*

*Proof.* Since no program flow belongs to *sof* bypasses $v$ (Lemma 1), no path can exist from $\ell$ to End that does not include $v$. Since all the paths from $\ell$ to End include $v$, $v$ post-dominates $\ell$ □

**Lemma 5** *Let $p$ be a label of a predicate such that $v$ post-dominates $p$, and there is a path from $p$ to $w$ that includes $v$. Then $w$ is not control dependent on $p$.*

*Proof.* There are two cases:

1. $w$ does not post-dominate $v$, so, the first condition in Def. 4 is negated, and it is not possible to make a control dependent relationship between $w$ and $p$.
2. $w$ post-dominates $v$. This causes $w$ to post-dominate $p$ as well. This post-domination negates the second condition in Def. 4.

Thus, in both cases, $w$ cannot be control dependent on $p$ □

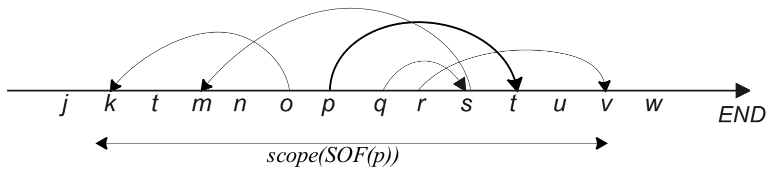


Fig. 9

**Lemma 6** *Let sof is an SOF, $p \in \mathcal{C}(sof)$, $\mathcal{I}(sof)=]k,v[$, and let $j$ be any label smaller than $k$. Then $j$ cannot be control dependent on $p$.*

*Proof.* Creating a forward path *pth* form $p$ to $j$ is mandatory to make $j$ control dependent on $p$. In accordance with Def. 4-1, $j$ must post-dominate all the labels in *pth* except $p$.
Since $j < p$ (the assumption of the lemma), *pth* is achieved by establishing a backward flow $\ell \rightarrow \ell'$ [8], where $\ell' \leq j$. We can divide *pth* into $pth_1$ and $pth_2$, where the first is from $p$ to $\ell$, and the second is from $\ell'$ to $i$. $pth_1$ requires to construct a chain of overlapping flows from $p$ to $\ell$. This chain could be formed by placing $\ell$ in one of the two intervals, in $\mathcal{I}(sof)$ where $k < \ell < v$ or in $v$ ahead where $v \leq \ell$.

- If $\ell \geq v$: since $v = Max(\mathcal{L}(sof))$ (Lemma assumption) and $v$ post-dominates $p$ (Lemma 4), all the paths from $p$ to $\ell$ include $v$. Accordingly, all the paths from $p$ to $j$ includes $v$ and, based on Lemma 5, $j$ cannot be control dependent on $p$.
- If $k < \ell < v$: $\ell \rightarrow \ell'$ indeed overlaps one of the flows in *sof*, and $\ell' \leq j$. As a result, $Min(\mathcal{L}(sof)) = j$ and not $k$. This is contrast to the assumption of the Lemma: $Min(\mathcal{L}(sof))=k$. Therefore, Label $j$ could not be control dependent on $p$ if $k < \ell < v$.

Since placing $\ell$ either in $\mathcal{I}(sof)$ or ahead of $v$ will not allow $p$ to control the execution of $j$. So, the lemma is proved. □

In Fig. 9, to establish a path from $p$ to $j$ that does not include $v$, we should create a backward flow $f_b$, wherein $k < f_b.out < v$ and $f_b.in \leq j$. In this case, $f_b$ bypasses $k$ and overlaps one of the flows in $SOF(p)$. As a result, $f_b$ is added to $SOF(p)$. Since $f_b.in < k$, $\mathcal{I}(SOF(p))=[f_b.in,v[$. This enlargement in the interval proves creating a control dependency between $p$ and $j$ is impossible.

**Lemma 7** *Let sof is an SOF, $p \in \mathcal{C}(sof)$, $\mathcal{I}(sof)=]k,v[$, and $w > v$. Then $w$ can not be control dependent on $p$.*

*Proof.* By Lemma 4, $v$ post-dominates $p$. Since $v$ post-dominates $p$ and $w > v$, all the paths from $p$ to $w$ includes $v$. As a result, and in accordance to Lemma 5, $w$ could not be control dependent on $p$. □

In Fig. 9, if we need $w$ to be control dependent on $p$, then we should add a flow $f$ from $\mathcal{I}(SOF(p))$ to any label less than $w$. Since $f.out \in \mathcal{I}(SOF(p))$ and $f.in > v$, $f$ will certainly overlap a flow in $SOF(p)$. As a consequence, $\mathcal{I}(SOF(p))$ will be enlarged, and $v \neq Max(\mathcal{L}(SOF(p)))$. Based on that, making any control dependence relationship between $p$ and any label larger than $v$ is impossible.

**Theorem 2.** *Let sof be an SOF, $p \in \mathcal{C}(sof)$, and $\mathcal{I}(sof)=]k,v[$. Then no possible control dependence relationship can be established between $p$ and a label outside $\mathcal{I}(sof)$.*

*Proof.* Lemma 6 states that it is not possible to establish a control dependence relationship between $p$ and label smaller than $k$. Lemma 7 states the same thing with labels larger than $v$. The two lemmas prove that $p$ can not control labels that exist outside $\mathcal{I}(sof)$. □

[8] This flow might also be a sequence of overlapping flows with interval from $\ell'$ to $\ell$. For the sake of simplicity, we consider it here as a one backward program flow.

### 3.4 Many SOFs

**Corollary 1.** *Let $\ell$ be a label of a statement. The set of predicates that control the execution of $\ell$ exist in $\mathcal{C}(\sum_{i=0}^{k} SOF_i(\ell))$.*

*Proof.* Theorem 2 states that the predicates could not control statements that do not belong to the interval of its SOF. Consequently, to make Label $\ell$ control dependent on the predicate $p$, both should exist in the same interval of SOF. □

**Lemma 8** *Suppose $f \in Jump(\ell)$ and $f.out$ has the largest output label among the outgoing labels in $\mathcal{L}(Jump(\ell))$. If sof is a SOF, and $f \in$ sof, then the predicates that control $\ell$ exist only sof.*

*Proof.* Since $f$ is a forward flow, $f.out < \ell < f.in$. If $f'$ is another forward flow bypassing $\ell$, there are two cases: the first is $f'.out < f.out < \ell < f'.in < f.in$, and in this case, there is an overlapping between $f$ and $f'$, and accordingly, $f'$ belongs to *sof*. Hence, the predicate controls $f'$ belongs to *sof*, which agrees well with the theorem.
The second case is $f'.out < f.out < \ell < f.in < f'.in$. If $p'$ controls $f'.out$, then it is impossible to form a path from one of its successors to $\ell$ wherein $\ell$ post-dominates all the labels due to the existence of $f$. Thus, the first condition of Def. 4 could not be satisfied, and $p'$ could not control the execution of $\ell$, and this proves the theorem. □

In Fig. 7, there are three SOFs, which are: $sof_1$={$f_1, f_2$}, $sof_2$={$f_3, f_4$}, $sof_3$={$f_5, f_6$}}. $\mathcal{C}(sof_1)$={3}, $\mathcal{C}(sof_2)$={6}, and $\mathcal{C}(sof_3)$={9}. $Jump(12)$= {$f_2, f_4$,$f_6$}. These three flows are forward. It is worth noting that the existence of $f_6$ prevents the predicates in $sof_1$ and $sof_2$ from satisfying the first condition in Def. 4. So, no path *pth* could be formed from one of the immediate successors of Label 3 to Label 12, wherein Label 12 post-dominates each label in *pth* except Label 3.

**Lemma 9** *Suppose many jump flows bypass Label $\ell$; each belongs to a different SOF. The flow $f$ is the only forward jump flow, whereas the remaining are backward jump flows. In assuming SOF($f$) = $SOF_m(\ell)$, the predicates that control $\ell$ exist in $SOF_m(\ell)$ and each $SOF_x(\ell)$ in $\sum_{i=0}^{q} SOF_i(\ell)$, whose $x < m$. In other words, the predicates that control $\ell$ exist in SOF($f$) and the other SOFs in $SOF_m(\ell)$ whose intervals are shorter than $\mathcal{I}(SOF(f))$.*

*Proof.* The ingoing label of the backward jump flow $f'$, whose SOF $sof'$ interval is less than $\mathcal{I}(SOF(f))$, falls between $f.out$ and $\ell$. Thereby, and contrary to the proof in Lemma 8, $f$ does not hinder forming a path *pth* from the predicates in $sof'$ to $\ell$, wherein $\ell$ post-dominates all the labels in *pth* except the predicate. So, by this, the first condition of Def. 4 could be satisfied.

The predicates in $SOF_x(\ell)$, where $x > m$, could not control $\ell$. The ingoing labels of their backward flows are less than $f.out$. So, by the proof of Lemma 8, it is not possible to create a path *pth* from their predicates to $\ell$ wherein $\ell$ post-dominate all the labels in *pth* except the predicate. □

The code of Fig. 8 exemplifies Lemma 9. It shows one forward jump flow: $6 \rightarrow 13$, and two backward jump flows: $12 \rightarrow 8$ and $16 \rightarrow 3$. Here, we have three SOFs, which are: $sof_1$={$5 \rightarrow 7, 6 \rightarrow 13$}, $sof_2$={$11 \rightarrow 13, 12 \rightarrow 8$}, $sof_3$= {$15 \rightarrow 17, 16 \rightarrow 3$}. It is worth noting that the three jump flows do not belong to one SOF. If Label 10 is the statement under analysis, which we need to find the predicates controlling it, we can say that predicates in $sof_2$ might control Label 10 because one of its backward flow $12 \rightarrow 8$ falls into 8, which is larger than 6. On the other hand, no predicate in $sof_3$ controls Label 10 because all its backward flows fall before Label 6.

Now, we can collect the results of these lemmas in one theorem, but before that, it is crucial to understand what the expression $\ell \in \mathcal{B}(sof)$ means. This expression means that all the flows in *sof* bypassing $\ell$ are backward.

**Theorem 3.** *Suppose* $genSOF(\ell) = \sum_{i=0}^{q} SOF_i(\ell)$*,* $\ell \notin \mathcal{B}(SOF_k(\ell))$*, and* $\forall x : x = 1$ *to* $k - 1, \ell \in \mathcal{B}(SOF_x(\ell))$*, where* $1 \leq k \leq q$*, the predicates that might control* $\ell$ *are in the set:* $\mathcal{C}(\sum_{i=0}^{k} SOF_i(\ell))$*.*

*Proof.* Corollary 1 determines that the predicates that might control any statement $\ell$ exit exclusively in $genSOF(\ell)$. Lemma 8 states that if $\ell$ is not in the backyard of many SOFs, then the predicates that might control $\ell$ exist only in the shortest SOF[9]. Lemma 9 states that if $\ell$ belongs to many backyard intervals in different SOFs, and *sof* is the shortest SOF in $genSOF(\ell)$, which $\ell$ does not belong to its backyard, then the predicates that control $\ell$ exist only in *sof* and other SOFs, whose intervals are shorter than *sof*. The theorem is proved. □

## 4 Super PCB (SPCB)

The majority of program flow overlapping occurs between structured and semi-structured flows and between structured and jump flows. These overlapping could be obtained immediately from the PCB graph itself, even though this requires enhancing the PCB graph to accept additional annotations.

This section introduces the "Super PCB" (SPCB) concept, a powerful tool for logically connecting PCBs. The SPCB, a collection of PCBs linked through semi-unstructured and jump flows, is more than just a concept. It's a game-changer. It not only implies overlapping flows in a single SOF but also eliminates the need for specific computations. This significant reduction in unnecessary computations underscores the value and importance of the SPCB concept.

This section first presents some necessary definitions, followed by rules logically connecting the PCBs in one SPCB.

### 4.1 Basic Definitions - Hierarchical Structure of the PCBs

**Definition 26.** *$\mathcal{P}$($pcb_i$) refers to the immediate parent of the PCB $pcb_i$.*

**Definition 27.** *$\mathcal{P}$($pcb_i$)=$pcb_j$ refers to the fact that the immediate parent of the PCB $pcb_i$ is the PCB $pcb_j$.*

The fact $\mathcal{P}(pcb_i)$=0 refers to the fact that $pcb_i$ is located in the main track, which means that it is not inside another conditional statement.

**Definition 28.** *$\top$($pcb_i$,$pcb_j$) refers to the common parent of the PCB $pcb_i$ and $pcb_j$.*

**Definition 29.** *$\top$($pcb_i$,$pcb_j$)= $pcb_b$ refers to the fact that the common parent of the PCB $pcb_i$ and $pcb_j$ is $pcb_b$.*

**Definition 30.** *$\overrightarrow{\mathcal{P}}$($pcb_i$) refers to the set of all the parents (outer conditional statements) of the PCB $pcb_i$.*

**Definition 31.** *$\overrightarrow{\mathcal{P}}$($pcb_i$,$pcb_j$) refers to a set of PCBs organized in a hierarchical structure from $pcb_i$ to $pcb_j$, wherein $pcb_i$ is the most inner PCB, while $pcb_j$ is the most outer PCB. $pcb_j$ is included in this set.*

[9] Shortest SOF means Shortest SOF Interval.

**Definition 32.** *$\overline{\mathcal{P}}(pcb_i,pcb_j)$ refers to a set of PCBs organized in a hierarchical structure from $pcb_i$ to $pcb_j$, wherein $pcb_i$ is the most inner PCB, while $pcb_j$ is the most outer PCB. $pcb_j$ is not included in this set.*

In Fig. 4, $\mathcal{P}(pcb_9)= pcb_5$. $\top(pcb_{14},pcb_{18}) = pcb_5$. $\overrightarrow{\mathcal{P}}(pcb_{10})=\{pcb_9, pcb_5, pcb_0\}$. $\overrightarrow{\mathcal{P}}(pcb_{10},pcb_5) = \{pcb_{10}, pcb_9, pcb_5\}$. $\overline{\mathcal{P}}(pcb_{10},pcb_5) = \{pcb_{10}, pcb_9\}$.

## 4.2 SPCB due to Semi-Structured Statement

The existence of a `continue` or `break` statement connects the PCBs as the following two rules:

**Rule 1** *Suppose there is a flow $\ell_1 \to \ell_2$, wherein $\ell_1$ is a label of a* `break` *statement and $\ell_1 \in pcb_1$ and $\ell_2 \in pcb_2$, then $\ell_1 \to \ell_2$ collects the set of PCBs $\overline{\mathcal{P}}(pcb_1,pcb_2)$ in a single SPCB.*

Fig. 10-a shows four conditional statements, each converted to a PCB. In naming each from the label of its predicate, it is noted that $\mathcal{P}(pcb_3)= pcb_1$, $\mathcal{P}(pcb_5)= pcb_3$, $\mathcal{P}(pcb_8)= pcb_5$, and $\mathcal{P}(pcb_8)= pcb_5$, and $\mathcal{P}(pcb_9)= pcb_8$. The flow of the `break` is $10 \to 15$, wherein $10 \in pcb_9$, and $15 \in pcb_1$. In accordance to Rule 1, this `break` collects $\overline{\mathcal{P}}(pcb_9,pcb_1)=\{pcb_9,pcb_8,pcb_5,pcb_3\}$ in a single SPCB.

**Rule 2** *Suppose there is a flow $\ell_1 \to \ell_2$, wherein $\ell_1$ is a label of a* `continue` *statement. If $\ell_1 \in pcb_1$, and $\ell_2 \in pcb_2$, then $\ell_1 \to \ell_2$ collects the PCBs $\overrightarrow{\mathcal{P}}(pcb_1,pcb_2)$ in a single SPCB.*

In Fig. 4 and in accordance to Rule 2, the `continue` statement at label 15, connects $P_{14}$ with its parent $P_5$ in one SPCB.

## 4.3 SPCB due to Jump Flows

The rule here shows how jump flow connects the PCBs in one SPCB.

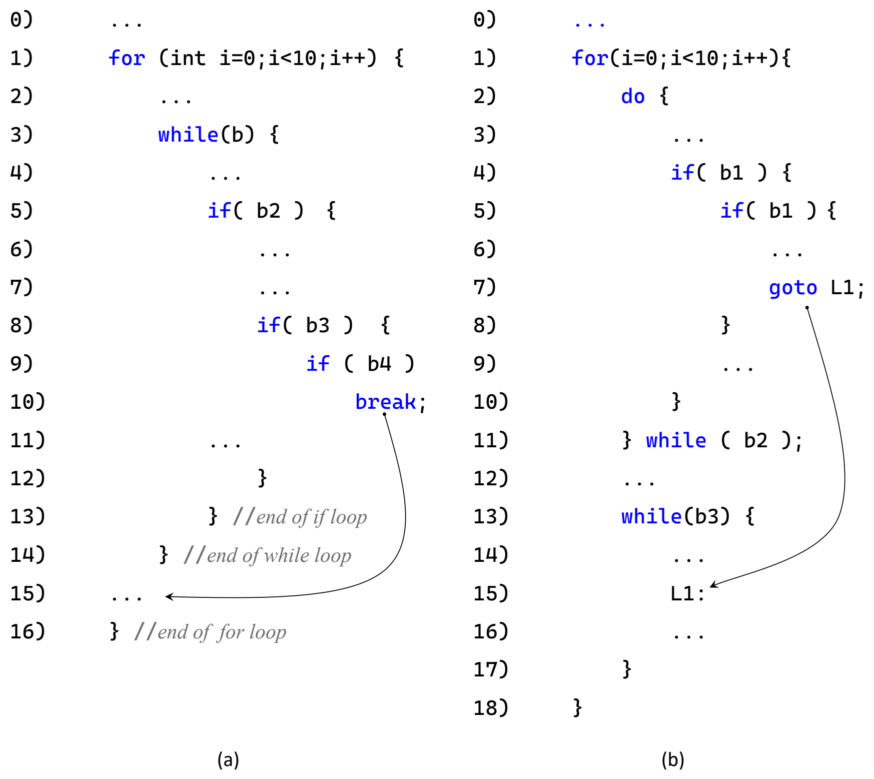
**Rule 3** *Suppose there are three PCBs; $pcb_i$, $pcb_j$ and $pcb_k$, wherein $\top(pcb_i,pcb_j)= pcb_k$, $\ell_x \in pcb_i$, $\ell_y \in pcb_j$, and $\ell_x \to \ell_y$ is a jump flow. Based on this assumption, an SPCB is formed from $\overline{\mathcal{P}}(pcb_i,pcb_k)$ and $\overline{\mathcal{P}}(pcb_j,pcb_k)$.*

The source code in Fig. 10-B has six PCBs, which are $pcb_0$, $pcb_1$, $pcb_2$, $pcb_4$, $pcb_5$, and $pcb_{13}$. The unstructured flow jumps from $pcb_5$ to $pcb_{13}$. $\top( pcb_5, pcb_{13})=pcb_1$. $\overline{\mathcal{P}}(pcb_5,pcb_1) = \{pcb_5, pcb_4, pcb_{11}\}$. $\overline{\mathcal{P}}(pcb_{13},pcb_1) = \{pcb_{13}\}$. Rule 3 forms a new SPCB from $\overline{\mathcal{P}}(pcb_5,pcb_1) + \overline{\mathcal{P}}(pcb_{13},pcb_1)=\{pcb_5, pcb_4, pcb_{11}, pcb_{13}\}$.

## 4.4 Augmented PCB Graph

In the PCB graph, PCBs are obtained in advance, whereas SPCBs could be computed on demand or in advance. Computing an SPCB requires the existence of some raw data or annotations in each PCB and in each flow.

– *sflow* is a set associated with the PCB. Suppose *pcb* is a PCB, then *pcb.sflow* includes each semi-structured flow whose outgoing label is inside *pcb*, or one of its internal PCBs, and its ingoing label is outside *pcb* or at *pcb.predicate*. Furthermore, this set includes each jump flow whose outgoing label is inside *pcb* and its ingoing label is outside the *pcb*, or whose outgoing label is outside *pcb* while its ingoing label is inside *pcb*.
– *pcbs* is a set associated with the program flow. Suppose $f$ is a program flow, then $f.pcb$ includes each PCB that $f$ launches from it or falls inside it.



```
...
for (int i=0;i<10;i++) {
    ...
    while(b) {
        ...
        if( b2 )  {
            ...
            ...
            if( b3 )  {
                if ( b4 )
                    break;
        ...
            }
        } //end of if loop
    } //end of while loop
...
} //end of for loop
```

(a)

```
...
for(i=0;i<10;i++){
    do {
        ...
        if( b1 ) {
            if( b1 ) {
                ...
                goto L1;
            }
            ...
        }
    } while ( b2 );
    ...
    while(b3) {
        ...
        L1:
        ...
    }
}
```

(b)

Fig. 10: Creating SPCB due to a `break` and `goto` statements

## 5 Forming on-the-fly SOFs from the CFG

The CFG represents program flows, while the SOF is a location type of information. Nevertheless, this section fills this gap by supposing that the label of the node in the CFG represents its location in the source code. It then uses these labels to assemble the required SOFs. Afterward, this section tries to push the performance to the highest possible point by improving the technique that finds the overlapping between program flows.

### 5.1 Algorithm 2: Basic Algorithm

Suppose that $F$ is a set of flows in a program, and its size is $N_f$. Finding the flows that bypass $\ell$ requires checking every $f$ in $F$ whether it bypasses $\ell$; if it is, then $f$ is added to a particular set (suppose *BypassingFlows*). Afterward, it forms the SOFs of these flows and adds them to $\sum SOF(\ell)$.

In Fig. 4, there are four SOFs which are: $\{f_1, f_2\}$, $\{f_6, f_7, f_8, f_9\}$, $\{f_3, f_4, f_{11}, f_{12}, f_{13}, f_{14}\}$, and finally $\{f_{15}\}$. To generate $\sum SOF(7)$, the algorithm collects first the flows bypassing Label 7 in the set *BypassingFlows*. Fig. 3 depicts that this set is: $\{f_2, f_3, f_4, f_5\}$. Second, the algorithm should fetch every flow $f$ from *BypassingFlows*, create a new SOF ($SOF(f)$), and add $f$ to it. Finally, it fetches every flow $f'$ from $SOF(f)$, finds every flow $f''$ in $F$ that overlaps with $f'$, adds $f''$ to $SOF(f)$, and recursively repeats this step many times until the algorithm can not add more flows to $SOF(f)$. Algorithm 2 shows this implementation.

**Algorithm 1:** Check the Overlapping Between Two Flows

```
1  Procedure CheckOverlapping(f,f´)
       Input:
       $f$ : program flow ;
       $f´$: program flow ;
2      if $f.out < f.in$ then
           // $f$ is a forward flow
3          if $f´.out > f.out$ AND $f´.out < f.in$ then
4              if $f´.in > f.in$ then return true ;
5              if $f´.in < f.out$ then return true;
6      else
           // $f$ is a backward flow
7          if $f´.out > f.in$ AND $f´.out < f.out$ then
8              if $f´.in > f.out$ then return true ;
9              if $f´.in < f.in$ then return true ;
10     if $f´.out < f´.ingiong$ then
           // $f'$ is a forward flow
11         if $f.out > f´.out$ AND $f.out < f´.in$ then
12             if $f.in > f´.in$ then return true;
13             if $f.in < f´.out$ then return true;
14     else
           // $f´$ is a backward flow
15         if $f.out > f´.in$ AND $f.out < f´.out$ then
16             if $f.in > f´.out$ then return true ;
17             if $f.in < f´.in$ then return true ;
18     return false
```

Algorithm 1 checks the overlapping between two flows $f$ and $f'$. Algorithm 2 builds on the fly the SOFs that bypass the statement $\ell$. The loop at Line 2 collects all the flows that bypass $\ell$ in one set (*BypassingFlows*). At Line 5, it fetches each flow $f$ in *BypassingFlow*, and checks the existence of *f.sof*. If it exists (was calculated before), then *f.sof* is added to $\sum SOF(\ell)$ (Line 7). Otherwise, the algorithm forms a new SOF *sof* (Line 9), and it adds $f$ to it (Line 10). Finally, the algorithm finds each flow overlaps with any flow in *sof* (Line 12), and adds it to *sof* (Line 15).

## 5.2 Rules for Improving the Performance of Algorithm 2

Algorithm 2 counts the values of edge labels in constructing $\sum SOF(\ell)$. This section places the flows (edges) into two arrays and uses four rules to accelerate the search operations in the array. Afterward, it implements the rules in two algorithms.

**Rule 4** *Suppose an array* `arr` *contains forward flows sorted in ascending order in terms of their outgoing labels, the size of* `arr` *is* `length`*, and the index of $\ell_d \to \ell_e$ in* `arr` *is $i$. If $\ell < \ell_d$, all the flows in* `arr` *whose indexes are from $i$ to* `length` *do not certainly bypass $\ell$.*

**Rule 5** *Suppose an array* `arr` *contains backward flows sorted in descending order in terms of their outgoing labels, the size of* `arr` *is* `length`*, and the index of $\ell_d \to \ell_e$ in* `arr` *is $i$. If $\ell > \ell_d$, all the flows in* `arr` *whose indexes are from $i$ to* `length` *do not certainly bypass $\ell$.*

**Algorithm 2:** Computing from CFG the SOFs that Bypass a Statement

```
1  Procedure BuildSOFsFromCFG($F$,$\ell$)
       Input:
       $F$: The set of program flows.
       $\ell$: The label of the statement under analysis.
       Data:
       $BypassingFlows$: The set saves the flows that bypass $\ell$.
       $sof$: a new SOF that will be added to $SSOF$
       Output:
       $\sum SOF(\ell)$ : the set of SOFs that bypass $\ell$.
2      foreach $f \in F$ do
3          if ($f.out < \ell$ AND $\ell < f.in$) OR ($f.in \leq \ell$ AND $\ell < f.out$) then
4              $BypassingFlows$ += $f$ ;
5      foreach $f \in BypassingFlows$ do
6          if $f.sof \neq null$ then
               // The SOF of $f$ is already computed
7              if $f$.sof $\notin \sum SOF(\ell)$ then $\sum SOF(\ell)$ += $f$.sof ;
8              continue ;
9          SOF $sof$ ;
10         $sof$ += $f$ ;
11         $f.sof = sof$ ;
12         foreach $f' \in F$ do
13             if $f'.sof \neq null$ then continue;
14             if CheckOverlapping($f$,$f'$) then
15                 $sof$ += $f'$ ;
16                 $f'.sof = sof$ ;
17         $\sum SOF(\ell)$ += $sof$ ;
18     return $\sum SOF(\ell)$
```

**Rule 6** *Suppose an array,* `arr`*, contains forward flows sorted in ascending order regarding their outgoing labels. The size of* `arr` *is* `length`*. The index of $\ell_d \to \ell_e$ in* `arr` *is $i$. There is a program flow $\ell_a \to \ell_b$. If $max(\ell_a,\ell_b) < \ell_d$, then all the flows whose indexes are from $i$ to* `length` *are indeed not overlapped with $\ell_a \to \ell_b$.*

**Rule 7** *Suppose an array,* `arr`*, contains backward flows sorted in an descending order regarding their outgoing labels. The size of* `arr` *is* `length`*. The index of $\ell_d \to \ell_e$ in* `arr` *is $i$. There is a program flow $\ell_a \to \ell_b$. If $min(\ell_a,\ell_b) > \ell_d$, then all the flows whose indexes are from $i$ to* `length` *are certainly not overlapped with $\ell_a \to \ell_b$.*

,

For example, if the array `arr` contains all the forward flows in Fig. 3 and they are sorted in ascending manner, then we get `arr`= $[f_1, f_2, f_3, f_6, f_8, f_9, f_{10}, f_{11}, f_{12}, f_{13}, f_{14}]$. From Rule 4, since $f_{11}.out > 17$, we conclude that the flows $\{f_{11}, f_{13}, f_{14}\}$ do not bypass label 17. From Rule 6, since $f_{13}.out$ is larger than $f_3.in$ and $f_3.out$, we conclude that $f_{13}$, and all the flows whose indexes are larger than the index of $f_{13}$ in `arr`, are not overlapped with $f_3$. These flows are: $\{f_{13}, f_{14}\}$.

**Algorithm 3:** Finding the Flows that Bypass $\ell$

```
1  Procedure ComputeBypassingFlows($\ell$, FWFlows, BKFlows )
       Input:
       $\ell$: a label under analysis.
       fwFlows: array of forward flows sorted.
       bkFlows: array of backward flows sorted.
       Data:
       f_fwrd: a forward flow that bypasses $\ell$.
       Output:
       BypassingFlows : the set of the flows that bypass $\ell$.
2      f_fwrd.out = -1 ;
3      foreach f in fwFlows do
4          if f.out $< \ell \wedge$ f.in $> \ell$ then
5              f_fwrd = f ;
6          if f.out $> \ell$ then break;
7      if f_fwrd.out> 0 then BypassingFlows += f_fwrd ;
8      foreach f in bkFlows do
9          if f.out $> \ell \wedge$ f.in $< \ell$ then
10             if f.in $\leq$ f_fwrd.out then continue;
11             BypassingFlows += f ;
12     if f.out $< \ell$ then break;
13     return BypassingFlows
```

### 5.3 Algorithm 3 (ComputeBypassingFlows)

Algorithm 3 precisely computes the flows that bypass the statement under analysis $\ell$, and stores them in *BypassigFlows*. Notably, Algorithm 3 distinguishes itself from Algorithm 2 by storing the flows in two distinct sets, *FWFlows* and *BKFlows*, a strategic implementation of the previously mentioned rules.

To implement Theorem 3, Algorithm 3 adds only one forward flow to *BypassingFlows* (Line 7). This forward flow, denoted as $f\_fwrd$, is selected because it has the maximum outgoing label among other forward flows bypassing $\ell$. Additionally, all the backward flows bypass $\ell$ and have ingoing labels greater than $f\_fwrd.out$ are added to *BypassingFlows* (Line 11). Without a forward flow bypassing $\ell$, all the backward flows bypassing $\ell$ are added to *BypassingFlows*.

### 5.4 Algorithm 4 (BuildSOFFromCFG2)

Algorithm 4 enhances Algorithm 2 to improve its performance concerning the execution times. The algorithm finds all the flows that bypass $\ell$ (Line 2) and collects them in *BypassingFlows*. Afterward, it fetches every flow $f$ from *BypassingFlows* to create its SOF *sof* to add it to $\sum SOF(\ell)$ (Line 18).

The algorithm creates $SOF(f)$ (Line 7), and add $f$ to it (Line 8). Then, it checks the overlapping between each flow $f$ in *sof* and every flow $f'$ in the program (Lines 10, 14) unless $SOF(f')$ was calculated before (Lines 11,15). If there is an overlapping between $f$ and $f\'$, $f\'$ is added to *sof* (Lines 12,16). The algorithm repeats the same steps with each flow in *sof*.

It is worth mentioning that the conditions at Lines 13, and 17 implement Rules 6 and 7 respectively.

**Algorithm 4:** Computing on-the-fly the SOFs that Bypass a Statement

```
1  Procedure BuildSOFFromCFG2($\ell$, fwFlows, bkFlows )
       Input:
       fwFlows: array of forward flows.
       bkFlows: array of backward flows.
       $\ell$: The label of the statement under analysis.
       Data:
       BypassingFlows: The set of the flows that bypass $\ell$.
       sof: a new SOF that will be added to $\sum SOF(\ell)$
       Output:
       $\sum SOF(\ell)$: the set of SOFs, which each has at least a flow bypassing $\ell$.
2      BypassingFlows = ComputeBypassingFlows($\ell$, fwFlows, bkFlows) ;
3      foreach $f \in$ BypassingFlows do
4          if $f.sof \neq null$ then
               // The SOF of $f$ is already computed
5              if $f.sof \notin \sum SOF(\ell)$ then $\sum SOF(\ell)$ += $f$.sof ;
6              continue ;
7          SOF sof ;
8          sof += $f$ ;
9          foreach $f \in$ sof do
10             foreach $f' \in$ fwFlows do
11                 if $f'.sof \neq null$ then continue ;
12                 if CheckOverlapping($f,f'$) then sof += $f'$ ;
13                 if $f'.out > f.out$ AND $f'.out > f.in$ then break ;
14             foreach $f' \in$ bkFlows do
15                 if $f'.sof \neq null$ then continue ;
16                 if CheckOverlapping($f,f'$) then sof += $f'$ ;
17                 if $f'.out < f.out$ AND $f'.out < f.in$ then break ;
18         $\sum SOF(\ell)$ += sof ;
19     return $\sum SOF(\ell)$
```

## 6 Creating SOF on-the-fly from SPCBs

The main aim of this work is to present an approach that forms SOFs on the fly. The previous section presents an approach built on the value of the labels, while this section presents an approach built on the PCB graph. This approach uses two algorithms to implement *genSOF*$(\ell)$. Selecting the most appropriate algorithm for each statement depends on the location of the statement. On top of them, Algorithm 7 selects the most appropriate algorithm for each statement.

### 6.1 Algorithm 5 (Core)

This algorithm is the internal engine for the procedure BuildSOFfromPCB. It gets a flow or a few flows constituting a subset of flows in an SOF; its role is to complete the new SOF. The subset of the flows reaches this algorithm through the parameter *stack*. Each flow $f$ is associated with a set called *pcbs*, which contains all the PCBs that $f$ overlap with their structured flows. In addition, each PCB *pcb* is associated with a set called *SFlows*, and this includes the structured flows of *pcb* as well as the semi-structured and jump flows that are overlapped with these structured flows. In brief, This algorithm gets the PCBs from the flows (Line 8) and the flows from the PCBs (Line 9), and after many iterations, it reaches most of the flows constituting the new SOF.

**Algorithm 5:**

```
1  Procedure CORE(stack)
       Input:
       stack : Stack of flows ;
       fwFlows: (Global Variable) array of sorted forward unstructured flows
       bkFlows: (Global Vairable) array of sorted backward unstructured flows
2      SOF sof ;
3      while stack.count > 0 do
4          f = stack.pop() ;
5          if $f \in$ sof then continue;
6          sof += f ;
7          f.sof = sof;
8          foreach p : f.pcbs do
9              foreach $f'$ : p.SFlows do stack.push($f'$);
10         if f.type IS NOT goto then continue ;
11         overlappingFlows = COLLECTOVERLAPPINGFLOWS(f,fwFlows,bkFlows) ;
12         foreach $f'$ : overlappingFlows do stack.push($f'$) ;
13     return sof ;
```

The set *SFlows* of the PCBs and *pcbs* of the flows could not recognize the overlapping between two jump flows. Thus, finding a jump flow $f$ in a *SFlows* set requires checking the overlapping between $f$ and each jump flow in the program (Line 11). If it finds one, then it adds it to *stack* (Line 12) to process it later as others.

In Fig. 3 and Fig. 4, $pcb_5.SFlows$=$\{f_3, f_4, f_5\}$, $pcb_{11}.SFlows$=$\{f_{11}, f_{12}\}$, $pcb_{14}.SFlows$=$\{f_5, f_{10}\}$ $pcb_{24}.SFlows$=$\{f_{13}, f_{14}\}$, $f_5.pcbs$=$\{pcb_5,pcb_{14}\}$, $f_{12}.pcbs$= $\{pcb_5,pcb_{11}\}$, and $f_{14}.pcbs$=$\{pcb_{13}\}$. Suppose BUILDSOFCORE is called and its *stack* = $\{f_{11}\}$, then BUILDSOFCORE recgonizes from $pcb_{11}.SFlows$ that $f_{12}$ should be added to *sof*. Since $f_{12}.pcbs$ contains $pcb_5$, then it adds all the flows of $pcb_5.SFlows$ to *sof*. From $f_5.pcbs$, the algorithm adds $f_{10}$ to *sof*, and since $f_{12}$ is a jump flow, it checks the overlapping between it and each jump flow in the program, thus, the algorithm adds Since $f_{14}$ to *sof*, and from $f_{14}.pcbs$, we get $f_{13}$. At the end, *sof*=$\{f_3, f_4, f_5, f_{10}, f_{11}, f_{12}, f_{13}, f_{14}\}$.

**Algorithm 6:**

```
1  Procedure BUILDSOFFROMPCB(pcb,$\ell$)
       Input:
       $\ell$: the label that we want to find its relevant SOFs.
       pcb: it is PCB of $\ell$.pcb or one of its parents.
       Data:
       $\sum SOF(\ell)$: The set of SOFs that bypass the $\ell$.
       stack : a stack of flows.
2      foreach f : pcb.SFlows do
3          stack.push(f)
4      $\sum SOF(\ell)$ += CORE(stack);
5      foreach f : pcb.InternalGOTOs do
6          if (f.out < $\ell$ AND $\ell$ < f.in) OR (f.in $\leq \ell$ AND $\ell$ < f.out) then
7              stack.clear(); stack.push(f);
8              $\sum SOF(\ell)$ += CORE(stack);
9      return $\sum SOF(\ell)$;
```

## 6.2 Algorithm 6 (BuildSOFfromPCB)

To build $SOF(pcb)$, the algorithm should add at least one of the flows in pcb.SFlows to *stack* variable, and then call CORE(*stack*)(Line 4).

The jump flows that are inside *pcb* and are not overlapped with the structured flows of $\ell.pcb$ are not stored *pcb.SFlows*. However, it is essential to recognize them because they form SOFs beneath $SOF(pcb)$, and most probably, their predicates control $\ell$. To distinguish them from the structured flows in *pcb.SFlows*, they are stored in another set called *pcb.InternalGOTOs*. Algorithm 6 considers these flows by finding the flows in *pcb.InternalGOTOs* that bypass $\ell$ (Line 6), and if there is any, it builds its SOF and adds it to $\sum SOF(\ell)$ (Line 8).

## 6.3 Algorithm 7 (GenerateSOFs)

**Algorithm 7:** The Top Procedure

```
1  Procedure GenerateSOFs($\ell$, $pcb$)
       Input:
       $sof$: An SOF that byapsses $\ell$ ;
       $pcb$: it is PCB of $\ell.pcb$ or one of its parents.
       Data:
       $\sum SOF(\ell)$: The set of SOFs which bypass $\ell$ ;
       // Case 1: generate the SOFs of $pcb$ with respect to $\ell$
2      if $pcb \neq pcb_0$ then
3          $\sum SOF(\ell)$ += BuildSOFfromPCB($\ell$,$pcb$)
       // Case 2: pcb is the main track, so, generates the SOFs from $Jump(\ell)$
4      if $pcb == pcb_0$ then
5          $bypassingFlows$ = ComputeBypassingFlows($\ell$,$fwFlows$,$bkFlows$) ;
6          foreach $f : bypassingFlows$ do
7              $\ell' = f.out$;
8              $\sum SOF(\ell)$ += GenerateSOFs($\ell'$, $\ell'.pcb$) ;
       // Case 3: $\ell$ exists in the backyard of $pcb$ or it does not belong $pcb$ interval.
9      else if $\ell \in \mathcal{B}(SOF(pcb))$ OR $\ell \notin \mathcal{I}(SOF(pcb))$ then
10         $\sum SOF(\ell)$ = GenerateSOFs($\ell$, $pcb.parent$)
11     return $\sum SOF(\ell)$
```

Algorithm GENERATESOFS implements $genSOF(\ell)$ and produces $\sum SOF(\ell)$. As it is shown in the details of Algorithm 7, there are three cases. In Case 1, the algorithm builds the SOF of *pcb*, which means it builds the SOF including the structured flows of *pcb* with respect to $\ell$. For this purpose, it calls the procedure BUILDSOFFROMPCB at Line 3. In Case 2, pcb is the main track and although the main track in the PCB graph is represented by a PCB but it does not have structured flows. Thus, Algorithm GENERATESOFS builds the SOFs of the jump flows bypassing $\ell$. Finally, Case 3 handles the situations when $\ell$ exit in the backyard of *pcb* or it does not belong to *pcb* interval. This means the predicate of the outer conditional statement might control its execution. Thus, the analysis raises up by building the SOFs in the parent PCB of *pcb*. It is worth noticing that Case 2 and 3 builds the new SOFs by calling recursively the procedure GENERATESOFS. At the end, all the SOFs are collects in $\sum SOF(\ell)$.

In Fig. 3, Labels 2, 9, 10, 26, and 28 belong respectively to the PCBs: $pcb_2$, $pcb_9$, $pcb_{10}$ $pcb_0$, and $pcb_{29}$, whose structured flows are: $f_1, f_6, f_8, null, f_{15}$. $SOF(pcb_2){=}SOF(f_1)$, $SOF(pcb_9){=}SOF(f_6)$, $SOF(pcb_{10}){=}SOF(f_8)$, and $SOF(pcb_{29}){=}SOF(f_{15})$.

In Algorithm 7, the condition of Case 2 is true for Label 26 because Label 26 $\in pcb_0$. The condition of Case 3 is true for Labels 2,9,26, and 28 because Label 2 $\notin \mathcal{I}(SOF(f_1))$, Label 9 $\in \mathcal{B}(SOF(f_6))$, and Label 28 $\in \mathcal{B}(f15)$.

**Algorithm 8:** Building SOF from a set of bypassingFlows

```
Procedure CollectOverlappingFlows($f$,$fwFlows$,$bkFlows$)
    Input:
    $f$: the flows that we want to find its SOF.
    $fwFlows$: array of sorted ascendingly forward goto flows w.r.t outgoing labels.
    $bkFlows$: array of sorted descendingly backward goto flows w.r.t. outgoing labels.
    Data:
    $overlappingFlows$: set of overlapping flows constitue SOF
    $stack$ : Stack of flows
    $stack$.push($f$) ;
    while $stack.NotEmpty()$ do
        $f$ = $stack$.pop() ;
        if $f$ is forward then
            foreach $f'$ in $fwFlows$ do
                if $f'.out > f.in$ then break ;
                if $f'.in \;\; f.out$ then continue;
                if $f'.out \;\; f.out \wedge f.out < f'.in \wedge f'.in > f.in \vee$
                    $f'.in > f.out \wedge f'.in < f.in \wedge f'.out < f.out$ then
                    if $f' \in overlappingFlows$ then continue ;
                    $overlappingFlows$.push($f'$); $stack$.push($f'$); continue ;
            foreach $f'$ in $bkFlows$ do
                if $f'.out < f.out$ then break;
                if $f'.in > f.in$ then continue;
                if $f'.out > f.out \wedge f'.out < f.in \wedge f'.in < f.out \vee$
                    $f'.in > f.out \wedge f'.in < f.in \wedge f'.out < f.out$ then
                    if $f' \in overlappingFlows$ then continue;
                    $overlappingFlows$.push($f'$); $stack$.push($f'$); continue ;
        else
            foreach $f'$ in $fwFlows$ do
                if $f'.out > f.out$ then break;
                if $f'.in < f.in$ then continue;
                if $f'.out > f.in \wedge f'.out < f.out \wedge f'.in > f.out \vee$
                $f'.out < f.in \wedge f'.in > f.in \wedge f'.in < f.out$ then
                    if $f' \in overlappingFlows$ then continue;
                    $overlappingFlows$.push($f'$); $stack$.push($f'$); continue ;
            foreach $f'$ in $bkFlows$ do
                if $f'.in < f.in$ then break;
                if $f.in > f.out$ then continue;
                if $f'.out > f.in \wedge f'.out < f.out \wedge f'.in \leq f.in \vee$
                $f'.in \geq f.in \wedge f'.in < f.out \wedge f'.out > f.out$ then
                    if $f' \in overlappingFlows$ then continue;
                    $overlappingFlows$.push($f'$); $stack$.push($f'$); continue ;
    return $overlappingFlows$ ;
```

### 6.4 Algorithm 8 (CollectOverlapping)

The flow $f$ is one of the input parameters to Algorithm 8 whose role is in computing $SOF(f)$. In addition to $f$, this algorithm has two input parameters. The first is *fwFlows*, which includes jump flows sorted in ascending manner concerning their outgoing labels. The second is *bkFlows*, which provides for jump flows sorted in ascending manner concerning their outgoing labels.

Algorithm 8 starts its logic by pushing $f$ into *stack*. Then, each flow in *stack* is popped as $f$ (Line 4), and all the flows that are overlapped with $f$ are stored temporarily in *stack* to find the flows that are overlapped with them.

The predicates at Lines 5, and 20 figures out the direction of $f$. This helps us in speeding the time required in testing the overlapping at Lines 10, 17, 25, and 32. The conditions at these lines consider the direction of $f$ and $f'$.

The algorithm impelments Theorem 3 by checking the overlapping between jump flows only. Further, it implements Rule 6 at Lines 7 and 14, wheres it implements Rule 7 at Lines 22, and 29.

## 7 Experimental Evaluations

This work implements the CFG-Based Approach (Algorithm 4) and PCB-Based Approach (Algorithm 7) that were developed by Microsoft Visual C++ 2022 (MVC). Programs are parsed using the "Regular Expression" built-in library in MVC, and the experiments have been performed on an Intel Core i7 13th Gen Intel(R) with a 2.1 GHz processor, 32 GB RAM, and 64-bit operating system. Both implementations run many times, and in each, the performance of both implementations is analyzed using a C file to measure mainly the time and space. The application gets the highest priority (Real-Time) and runs with an affinity processor.

### 7.1 Varying the number of the Flows

|  | No.Flows | No.Predicates | No. break& continue statements | No.Jumps | SPCB Time (ms) | CFG Time (ms) | Speed-Up | SPCB Space | CFG Space | Space Consumption | No. SOFs |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2K | 1,165 | 616 | 279 | 2 | 0.6 | 1.9 | 3.2 | 944 | 820 | -1.15 | 299 |
| 10K | 5,799 | 3,103 | 1,389 | 10 | 3.5 | 40 | 11 | 3,040 | 2,100 | -1.45 | 1,481 |
| 25K | 14,411 | 7,647 | 3,370 | 24 | 6.1 | 176 | 30 | 6,612 | 4,460 | -1.48 | 3,688 |
| 50K | 28,971 | 15,404 | 6,827 | 50 | 14.2 | 669 | 47 | 12,792 | 8,808 | -1.45 | 7,396 |
| 75K | 42,990 | 22,835 | 10,057 | 74 | 21 | 1,457 | 69 | 18,664 | 12,240 | -1.52 | 11,126 |
| 100K | 56,805 | 30,196 | 13,242 | 100 | 23 | 2,565 | 111 | 24,448 | 16,036 | -1.52 | 14,675 |
| 150K | 84,879 | 45,140 | 19,842 | 150 | 38.6 | 5,830 | 151 | 36,048 | 23,640 | -1.52 | 21,979 |
| 200K | 112,237 | 59,665 | 25,940 | 200 | 53 | 10,237 | 193 | 47,768 | 31,184 | -1.53 | 29,372 |

Table 1: Results from Experimental Evaluations

The performance of any approach computing the SOFs on demand depends on the number of flows. To study this factor, a code generator created eight C source code programs. The files vary their number of statements and, consequently, their number of program flows. Afterward, a comparison is held to understand the difference in their performances.

Table 1 summarizes the results of our measurements. The first column shows the synthetic files, which vary from 2K to 200K. The second column shows the number of the total program flows in each file. The third and fourth columns show the number of predicates and the number of semi-structured flows in each file, respectively. The columns SPCB time and CFG Time show the execution time needed to create all the SOFs of each file using the two approaches. Then, the Speedup column shows the speedups obtained using the "SPCB method" over the "CFG method." We also compare the memory consumption of the two approaches methods in this table. The Save-up shows how much we save from using the PCB-graph-based approach over the CFG-based approach.

## 7.2 Number of `goto` statements

Algorithm 5 at Line 11 might degrade the performance of the PCB-based approach because it checks the overlapping between every jump flow added recently to a new SOF with each jump flow in the program. Thus, studying the effect of the number of jump flows is crucial. For measurement purposes, we generated C program files with the same sizes and whose number of `goto` statements varies from 120 to 5400. Then, both implementations, the PCB-based and CFG-based graphs, run on each of these files, and the measurements are recorded in Table 2.

| | No.Predicates | No.breaks&continue | No.Jumps | SPCB Time (ms) | CFG Time (ms) | Speed-Up | SPCB Space (K.B.) | CFG Space (K.B.) | Space Save Up | No. SOFs |
|---|---|---|---|---|---|---|---|---|---|---|
| 120K120goto | 34,923 | 14,625 | 120 | 24.02 | 2,786 | 116.0 | 28,208 | 19,000 | -1.48 | 17,764 |
| 120K1800goto | 34,698 | 12,764 | 1,800 | 35.33 | 2,755 | 78.0 | 29,608 | 18,714 | -1.58 | 16,961 |
| 120K3600goto | 34,818 | 11,063 | 3,600 | 73.21 | 3,405 | 46.5 | 31,280 | 18,884 | -1.66 | 15,919 |
| 120K5400goto | 34,712 | 9,293 | 5,396 | 99.70 | 2,755 | 27.6 | 32,224 | 28,280 | -1.14 | 14,790 |

Table 2: Studying the effect of varying the number of `goto` statements

The experiments in Table 2 evaluate four automatic-generated files. The observations show that the execution time with the CFG-based approach has not been affected, but it has increased four times with the PCB-based approach.

## 8 Discussions

The experimental evaluations compare two approaches; each can form the SOFs of a particular statement on the fly. The first approach builds its algorithms on the CFG, while the second builds its algorithms on the PCB graph. The goal of presenting the first approach is to use it as a baseline in the evaluations. It is worth noting that each experiment produces all the SOFs in each C file, although both are designed for on-demand computations.

The observations in Table 1 show that the first approach (Algorithm 4) is sensitive to the increase in program flows. Table 1 depicts that increasing the number of program flows 100 times (from 1,165 to 112,237) increases the execution times 5,000 times (from 1.9 m.s. to 10,237 m.s.). The cause of this unscalability is the quadratic time complexity in forming the SOFs. If $N_f$ is the number of program flows, and adding a new program flow $f$ to a SOF needs checking the overlap between $f$ and each program flow, then the time complexity is $O(N_f^2)$.

Table 1 shows that increasing the number of program flows 100 times (from 1,165 to 112,237) increases the execution times of the PCB-based approach 100 times. The execution times do not exceed 53 m.s., which is fast. These results demonstrate a high-performance, consistent, and scalable approach. The favor for this is the PCB graph's structure, which enables immediate capturing of the structured flows that bypass a particular statement with their relevant semi-structured flows.

The question is, why does using PCB graphs and its enhanced SPCB version give this superiority? The answer is simple because they have information types that do not correspond with CFGs. Suppose the statement under analysis is $\ell$; the PCB graph can immediately figure out whether $\ell$ is inside a conditional statement, what is the structured flow(s) bypassing $\ell$, and what are the semi-structured and jump flows that are related to these structured flows. The immediate capturing of such information from the program representation eliminates the need to make an expensive search between the program flows.

Another question comes to mind: What is the price of adding annotations to the PCB graph to make it able to build the SPCBs immediately on demand, thereby getting these scalable and fast manipulations? The answer is in the memory save-up column in Table 1. The trade-off between space and speed requires adding a memory space of 1.5. This price is cheap.

The PCB-based approach's main challenge is the number of `goto` statements. If $pcb_0$ (main track) is very long and there are many `goto` statements, then finding the flows that bypass the statement under analysis $\ell$, which belongs to $pcb_0$, requires individually checking the jump flows in the main track. Further, the time complexity for forming the overlapping flows is $O(n_j^2)$, where $n_j$ is the number of jump flows in $pcb_0$. This case also occurs with any conditional statement having many jump flows, although this is rare.

Table 2 studies the effect of increasing the number of jump flows. The data shows that varying the number of jump flows from 120 to 5400 increases the execution time from 24 m.s. to 99.7 m.s. In other words, increasing the number of `goto` statements 45 times leads to increasing the execution time four times. Nevertheless, the PCB-based approach is still scalable with the number of jump flows.

The percentage of `goto` statements in the files of Table 2 reaches 4.5%. This percentage is very rare. The empirical study of Meiyappan Nagappan et el [7]. found that 88.5% of files among 2,150,387 files do not have any `goto` statement, 14.45% of the files containing goto statements have only one `goto` statements, and the percentage of the files that has more than 100 goto statements is 0.55%. Based on that, most of the flows are either structured or semi-structured. As a result, checking the overlapping between unstructured flows does not cause a bad performance. However, the experiments show that the performance of the PCB-based approach is good with a large number of `goto` statements.

## 9 Related Work & History

For more than four decades, researchers in static program analysis have been working on the control dependence research question as one of the requirements for program slicing. In 1981, Weiser [8] introduced the program slicing concept. He used dataflow equations for this purpose, but his method needed to be more appropriate. It does not work with unstructured programs. Ottenstein and Ottenstein [5] invented the Program Dependence Graph (PDG) for finding slices in structured programs, but PDG-based slicing cannot handle such dependencies in unstructured programs.

In the PDG-based slicing, the control dependencies are obtained from the post-dominator tree [5]. Many algorithms arose [9,10,11,12,13,14,15,16] to collect the post-domination facts in the program and then arrange them in dominator or post-dominator trees. All these algorithms are comprehensive, meaning for computing the post-domination for Node $n_i$, the algorithm should adequately calculate the post-domination for either the predecessors or successors of $n_i$. The most famous dominator algorithm is the one provided by Lengauer-Tarjan's [15].

Finding the control dependencies of unstructured programs was one of the main challenges in program slicing for about two decades. Many works [17,18,19] tried to find a solution. Ball and Horwitz [19] as well as Choi and Ferrentai [18] proposed Augmented CFG (ACFG) and Augmented PDG (APDG). They treated the `goto` statements as predicates. Harman and Danicic extended Agrawal's algorithm and performed better using a refined criterion for slicing the `goto` statements. Sinha et al. [20] mentioned that the Harman and Danicic algorithm needs to be more precise with `switch` statements, and she proposed a solution for inter-procedural control dependencies.

Demand-driven slicing aims to avoid making comprehensive analyses to get the facts for one or two statements that might be among tens or hundreds of thousands of source code lines. These approaches were presented by Kraft [21], Sandberg et al. [22], Lisper et al. [23], Atkinson and Griswold [24,25,26,27], and Hajnal and Forgács [28]. It is worth noting that none of these works proposed a way to obtain the control dependencies in unstructured programs on the fly.

Our previous work in 2015 [1] proposed the new program representation Predicated Code Block (PCB) graph as an alternative to the Control Flow Graph (CFG). Upon this new representation, the work [1] builds a demand-driven slicing approach for intra-procedural structured programs. The following work in 2016 [2] enhanced the PCB graph to handle inter-procedural cases. Afterward, our work in 2019 [3] invented a new method that computes the control dependencies in unstructured programs on the fly. This novel work operates in two distinct steps. The first gets the predicates that control the statement under analysis from an SOF. This step is over-approximated but fast. The second stage performs a deep analysis to pick the minimum predicates. The challenge of this approach is the complexity of the time spent building the SOFs. This paper fixes this problem.

## 10 Conclusions

The Predicated Control Block (PCB) is a novel program representation that encapsulates the representations of conditional statements, the syntactic structure of conditional statements (child-parent relations), the location information of statements, the type of conditional statements (Linear or Cycle), and the program flows. This paper introduces a new dimension to the PCB graph by representing the logical connections between the PCBs. These logical connections can be understood as the relationships between different conditional statements within the program. Furthermore, it added the power of classifying the program flows into structured, semi-structured, and jump flows.

Contrary to the PCB graph, the Control Flow Graph, the state-of-the-art representation, represents only control flows and neglects other types of information. Focusing on the representation of one type of information and neglecting others causes the emergence of approaches that sometimes try to retrieve the information that is distracted from the available information indirectly. The best example for this is the control dependence. Each statement is controlled and dependent on the predicate of its conditional statement unless some other circumstances occur. This simple concept illustrates how the eye reflects when it looks at the code. Neglecting this simple fact yields to make every predicate suspected to control the statement under analysis.

This work and its predecessors [1,2,3,4] show one rule; *more types of information represented, more effortless analysis*. This way of working enables us to develop high-performance, on-demand, scalable, and accurate analysis. More importantly, it prevents the need to develop comprehensive analysis.

## References

1. Husni Khanfar, Björn Lisper, and Abu Naser Masud. Static backward program slicing for safety-critical systems. In *Ada-Europe International Conference on Reliable Software Technologies*, pages 50–65. Springer, 2015.
2. Husni Khanfar and Björn Lisper. Enhanced PCB-based slicing. In *Fifth International Valentin Turchin Workshop on Metacomputation*, page 71, 2016.
3. Husni Khanfar, Björn Lisper, and Saad Mubeen. Demand-driven static backward slicing for unstructured programs. Technical Report MDH-MRTC-324/2019-1-SE, School of Innovation, Design and Engineering. Malardalen University, May 2019.
4. Husni Khanfar. Computing on-the-fly the relevant program flows to a control dependency. Technical Report MDH-MRTC-334/2021-1-SE, School of Innovation, Design and Engineering. Malardalen University, April 2021.
5. Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, volume 19, pages 177–184. ACM, 1984.
6. Venkatesh Prasad Ranganath, Torben Amtoft, Anindya Banerjee, John Hatcliff, and Matthew B Dwyer. A new foundation for control dependence and slicing for modern program structures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):27, 2007.
7. Meiyappan Nagappan, Romain Robbes, Yasutaka Kamei, Éric Tanter, Shane McIntosh, Audris Mockus, and Ahmed E Hassan. An empirical study of goto in c code from github repositories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 404–414, 2015.
8. Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
9. Alfred V Aho and Jeffrey D Ullman. *Principles of Compiler Design (Addison-Wesley series in computer science and information processing)*. Addison-Wesley Longman Publishing Co., Inc., 1977.
10. Alfred V Aho and Jeffery D Ullman. Lr (k) grammars. In *The theory of parsing, translation, and compiling*, volume 1, pages 371–379. Prentice-Hall Englewood Cliffs, NJ, 1972.
11. Stephen Alstrup, Dov Harel, Peter W Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.
12. Adam L Buchsbaum, Haim Kaplan, Anne Rogers, and Jeffery R Westbrook. Linear-time pointer-machine algorithms for least common ancestors, mst verification, and dominators. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 279–288. ACM, 1998.
13. Matthew S Hecht. *Flow analysis of computer programs.* Elsevier Science Inc., 1977.
14. Matthew S Hecht and Jeffrey D Ullman. A simple algorithm for global data flow analysis problems. *SIAM Journal on Computing*, 4(4):519–532, 1975.
15. Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(1):121–141, 1979.
16. Paul W Purdom Jr and Edward F Moore. Immediate predominators in a directed graph [h]. *Communications of the ACM*, 15(8):777–778, 1972.
17. Hiralal Agrawal. On slicing programs with jump statements. In *ACM Sigplan Notices*, volume 29, pages 302–312. ACM, 1994.
18. Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1097–1113, 1994.

19. Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *International Workshop on Automated and Algorithmic Debugging*, pages 206–222. Springer, 1993.
20. Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 432–441. IEEE, 1999.
21. Johan Kraft. *Enabling timing analysis of complex embedded software systems*. PhD thesis, Mälardalen University, 2010.
22. Christer Sandberg, Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Faster wcet flow analysis by program slicing. In *ACM SIGPLAN Notices*, volume 41, pages 103–112. ACM, 2006.
23. Björn Lisper, Abu Naser Masud, and Husni Khanfar. Static backward demand-driven slicing. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation*, pages 115–126. ACM, 2015.
24. Darren C Atkinson and William G Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 52. IEEE Computer Society, 2001.
25. Leeann Bent, D Atkinson, and W Griswold. A qualitative study of two whole-program slicers for c. *Technical Report*, 2000.
26. Darren C Atkinson and William G Griswold. Effective whole-program analysis in the presence of pointers. *ACM SIGSOFT Software Engineering Notes*, 23(6):46–55, 1998.
27. Darren C Atkinson and William G Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th international conference on Software engineering*, pages 16–27. IEEE Computer Society, 1996.
28. Ákos Hajnal and István Forgács. A demand-driven approach to slicing legacy cobol systems. *Journal of software: evolution and process*, 24(1):67–82, 2012.