Demand-Driven Static Backward Slicing for Unstructured Programs

Husni Khanfar, Björn Lisper, Saad Mubeen

School of Innovation, Design, and Engineering, Mälardalen University,
SE-721 23 Västerås, Sweden
husni.khanfar@mdh.se, bjorn.lisper@mdh.se, saad.mubeen@mdh.se

Abstract. Backward program slicing identifies the program parts that might influence a particular variable at a program point. A program part (e.g., a statement) can be directly influenced by another part due to its data or control dependence on the later. The classical program slicing approaches are designed to find in advance all the data and control dependencies in the program. This design entails a considerable amount of unnecessary computations because not all the dependencies are required for computing the slice. Demand-driven program slicing approaches try to raise the analysis performance by avoiding the unnecessary computations. However, these approaches cannot address unstructured programs in a demand-driven fashion. On the other hand, the existing techniques that compute the control dependencies in unstructured programs are based on fixed-point iterations, which limits their integration to the demand-driven slicing approaches.
Program slicing based on Predicate Code Block (PCB) is a new demand-driven slicing approach that can address only structured programs. This paper presents the first demand-driven technique to compute the control dependencies in unstructured programs. In this regard, the technique uses flow information, location-based information and syntactic structure of the source code. Further, the paper shows how the new technique can be integrated to the PCB-based slicing approach to address unstructured programs.

Keywords: Program Analysis, Predicate Control Block, Control Dependence, Slicing, Unstructured Programs.

1 Introduction

In program analysis techniques, many problems are characterised as problems of scale due to the sheer size of some large programs. The size of such programs could be reduced by using program slicing. This technique reduces program size concerning a *slicing criterion*, which is a pair $< var, loc >$, where *var* is a variable at the program point *loc*. The slicing techniques aim at removing the program parts that do not affect directly or indirectly the slicing criterion. The remaining parts constitute a *slice*, from the original program P, whereby the *slice* computes the same values of *var* at *loc* to those obtained by P, whenever both P and *slice*

have the same program inputs. The different parts of a program can affect each other through data or control dependencies. A statement s_2 is data dependent on another statement s_1 if s_1 assigns a value to a variable that is read by s_2. Similarly, a program statement s is control dependent on predicate b if the value of b determines whether s is executed. Program slicing has potential use in many areas such as program debugging, verification, testing and maintenance.

PCB-based slicing [1,2] is a recently developed program slicing approach. This approach is demand driven in the sense that it computes only necessary dependencies instead of computing all possible dependencies within a program. As a result, the PCB-based slicing outperforms the slicing approach that was developed by Ottenstein and Ottenstein [3], and is based on Program Dependence Graph (PDG). Further, this approach operates on a new light-weight program representation that is referred to as the *PCB graph*. The PCB-graph represents the program flows. Moreover, the PCB-graph preserves the syntactic structure of program. This mix of two types of information is contrary to the concept applied in the state-of-the-art program representation, namely the Control Flow Graph (CFG), which focuses only on representing the program flows.

The PCB-based slicing approach shown in [1,2] works well with structured programs, but it does not address unstructured programs. The difference between slicing the structured and unstructured programs is in computing the control dependencies in the presence of unstructured jumps (e.g. goto, break) in the case of latter. This report aims at covering this gap by designing a novel technique to compute on the fly[1] the control dependencies in unstructured programs.

In the state-of-the-art slicing approaches, the control dependence relationship is obtained from the post-domination facts[2]. These facts are obtained by some classical methods [4,5]. These methods are not demand-driven because they are based on fixed-point iterations, causing the computation of unnecessary information.
The main contributions presented in this paper are:

1. developing a new approach based on the PCB-graph to compute the control dependencies in unstructured programs,
2. extending the PCB-based slicing approach to address unstructured programs.

The first contribution is based on three theorems. This contribution is carried out in two phases. The first phase (based on Theorem 1 and Theorem 3) is quick, safe, but approximate. It quickly removes many control-dependence candidates. The second phase (based on Theorem 2) is exact but potentially expensive. The idea is that if the first phase manages to screen many potential control dependencies, then there will be only a few dependencies left to be investigated with the second phase. To sketch the three theorems out, given a statement s that exists inside a conditional statement whose predicate is p:

[1] The terms in *demand-driven fashion* and *on-the-fly* are used interchangeably here.
[2] The post-domination fact is defined in Section 2.3

1. Theorem 1 can be implemented to exclude the predicates that certainly do not control s,
2. Theorem 2 supports a check whether there is a standard control dependency between a given statement and a given predicate,
3. Theorem 3 supports a check whether s is control dependent on predicates other than p.

It is worth mentioning that the proposed approach for computing the control dependencies in unstructured programs mainly targets high-level programs that are "almost well-structured". In other words, they have only a few unstructured jumps, as opposed to completely unstructured programs without structured program constructs like binary code with jumps only.

The rest of the paper is organized as follows, Section 2 provides the background. Section 3 discusses the types of program flows and some behaviors related to their locations. Section 4 discusses the approach to compute on the fly the control dependencies. Section 5 optimizes the approach proposed in Section 4. Section 6 presents the formal definition of the PCB-graph representation. Section 7 provides a complete slicing algorithm based on the PCB graph. Section 8 presents the related work and Section 9 provides the conclusions.

2 Background

This section provides a brief description of the While language, the state-of-the-art program representation CFG, the post-domination concept and the control dependencies.

2.1 While Language

The While language [6] is a small model imperative programming language. This language is used to develop and test new approaches and methods specialized in the analysis of source codes. Using the While language for such developments gets rid of tons of details included in the real languages. The details are certainly not needed in the theoretical development.

A While program is a statement s, which might be an elementary statement (es), conditional statement (cs) or a composite statement ($s_1; s_2$). In [6], every elementary statement or a predicate has a unique integer label. This report extends the labeling scheme in [6] by giving a unique label to every conditional statement. Further, a `goto` statement is added to the syntax of the While language. The statements are labeled in ascending order according to their locations in the source code, from left to right and from top to bottom. With this labeling system, all program flows except backward jumps go from statements with lower labels to statements with higher labels. This work extends the While language presented in [6] to include the `goto` statement.

Let a denote arithmetic expressions, and the predicate b denotes boolean expressions. The abstract syntax of the While language is:

$$
\begin{aligned}
cs &::= \texttt{if } [b]^{\ell} \texttt{ then } s' \;\mid\; \texttt{if } [b]^{\ell} \texttt{ then } s' \texttt{ else } s'' \;\mid\; \texttt{while } [b]^{\ell} \texttt{ do } s' \\
es &::= [x := a]^{\ell} \;\mid\; [skip]^{\ell} \;\mid\; [goto\ \ell']^{\ell} \\
s &::= es \;\mid\; s'; s'' \;\mid\; cs
\end{aligned}
$$

If clear from the context, we will abuse notation and write "predicate p", or "statement s" instead of "the label of predicate p" or "the label of statement s".

2.2 Building a CFG from a While Program

The 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, wherein each node represents a predicate or an elementary statement, and each edge represents a possible program flow. The node is the label. The control flow edges in the CFG are formed from a pair of labels (i, j), which means that j might be executed after i.

In building a CFG for a While program, there is a control flow edge from each elementary statement to its immediate successor. Furthermore, there are two control flow edges from every conditional statement to its immediate successors. The internal flows in every conditional statement are determined based

on the internal structure of this conditional statement. If we suppose that cs is a `while` conditional statement, cs comprises a predicate and a body. There are two flows from the predicate of cs, the first is a flow from the predicate to the first statement in the body of cs and the second is from the predicate of cs to the immediate successor of cs. In addition, there is a control flow from the last statement in the body of cs to its predicate. If cs' is an `if` conditional statement, cs' comprises a predicate and a body. There are two control flow edges from the predicate of cs'. The first edge is from the predicate to the first statement in the body of cs', and the other is from the predicate to the immediate successor of cs'. Further, there is another flow from the last statement in the body of cs' to the immediate successor of cs'. In assuming that cs'' is an if-else conditional statement, cs'' is composed of a predicate and two bodies. There is a control flow edge from the predicate of cs'' to the first statement in each body. In addition, there is a control flow edge from the last statement in each body to the immediate successor of cs''. We refer the reader to the book "Principles of Program Analysis" [6] for further details about the construction of the CFG from a While program. The formal definition of building a CFG from a While program in [6] defines five types of functions: *init, final, blocks, labels* and *flow* to construct a CFG from a While program. In these definitions, Lab refers to the set of all labels in the program. These definitions are extended to include `goto` statements as follows:

$$init : \text{Stmt} \rightarrow \text{Lab}$$

which gets the initial label of a statement:

$$\begin{aligned}
init([x := a]^\ell) &= \ell \\
init([skip]^\ell) &= \ell \\
init([goto\ \ell']^\ell) &= \ell \\
init(\text{S}_1; \text{S}_2) &= init(\text{S}_1) \\
init(\text{if } [b]^\ell \text{ then S}) &= \ell \\
init(\text{while } [b]^\ell \text{ do S}) &= \ell \\
init(\text{if}[b]^\ell \text{ then S}_1 \text{ else S}_2) &= \ell
\end{aligned}$$

$$final : \text{Stmt} \rightarrow \text{P(Lab)}$$

which gets the set of last labels in a statement:

$$
\begin{aligned}
final([x := a]^\ell) &= \{\ell\} \\
final([skip]^\ell) &= \{\ell\} \\
final([goto\ \ell']^\ell) &= \{\ell\} \\
final(\mathrm{S}_1; \mathrm{S}_2) &= final(\mathrm{S}_2) \\
final(\text{if } [b]^\ell \text{ then S}) &= \{\ell\} \cup final(\mathrm{S}) \\
final(\text{while } [b]^\ell \text{ do S}) &= \{\ell\} \\
final(\text{if}[b]^\ell \text{ then } \mathrm{S}_1 \text{ else } \mathrm{S}_2) &= final(\mathrm{S}_1) \cup final(\mathrm{S}_2)
\end{aligned}
$$

$$blocks : \mathrm{Stmt} \rightarrow P(Blocks)$$

where *Blocks* is the set of elementary statements and predicates [6]. It is defined as follows:

$$
\begin{aligned}
blocks([x := a]^\ell) &= \{[x := a]^\ell\} \\
blocks([skip]^\ell) &= \{[skip]^\ell\} \\
blocks([goto\ \ell']^\ell) &= \{[goto\ \ell']^\ell\} \\
blocks(\mathrm{S}_1; \mathrm{S}_2) &= blocks(\mathrm{S}_1) \cup blocks(\mathrm{S}_2) \\
blocks(\text{if } [b]^\ell \text{ then S}) &= \{[b]^\ell\} \cup blocks(\mathrm{S}) \\
blocks(\text{while } [b]^\ell \text{ do S}) &= \{[b]^\ell\} \cup blocks(\mathrm{S}) \\
blocks(\text{if}[b]^\ell \text{ then } \mathrm{S}_1 \text{ else } \mathrm{S}_2) &= \{[b]^\ell\} \cup blocks(\mathrm{S}_1) \cup blocks(\mathrm{S}_2)
\end{aligned}
$$

$$labels : \mathrm{Stmt} \rightarrow \rho(Lab)$$

$$labels(S) = \{\ell | [B]^\ell \in Blocks(S)\}$$

$$flow : \mathrm{Stmt} \rightarrow \mathrm{P}(\mathrm{Lab} \times \mathrm{Lab})$$

$$
\begin{aligned}
flow([x := a]^\ell) &= \emptyset \\
flow([skip]^\ell) &= \emptyset \\
flow([goto\ \ell']^\ell) &= (\ell, \ell') \\
flow(\mathrm{S}_1; \mathrm{S}_2) &= flow(\mathrm{S}_1) \cup flow(\mathrm{S}_2) \\
&\cup \{(\ell, init(\mathrm{S}_2)) | \ell \in final(\mathrm{S}_1) \wedge \ell \text{ is not a label of a goto statement}\} \\
flow(\text{if } [b]^\ell \text{ do S}) &= flow(\mathrm{S}) \cup \{(\ell, init(\mathrm{S}))\} \\
flow(\text{while } [b]^\ell \text{ do S}) &= flow(\mathrm{S}) \cup \{(\ell, init(\mathrm{S}))\} \cup \{(\ell', \ell) | \ell' \in final(\mathrm{S})\} \\
flow(\text{if}[b]^\ell \text{ then } \mathrm{S}_1 \text{ else } \mathrm{S}_2) &= flow(\mathrm{S}_1) \cup flow(\mathrm{S}_2) \cup \{(\ell, init(\mathrm{S}_1)), (\ell, init(\mathrm{S}_2))\}
\end{aligned}
$$

Definition 1. ***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.*

2.3 Post-domination and Control Dependence

Definition 2. ***Post-domination:*** *In a CFG G, any node n* post-dominates *node y if all the paths from y to Exit contain n.*

Definition 3. ***Strict Post-dominator:*** *In a CFG, a node n* strictly post-dominates *node y if n post-dominates y and n is not equal to y.*

Definition 4. ***Immediate Post-dominator:*** *In a CFG, the node n* immediately post-dominates *another node m if n strictly post-dominates m but does not strictly post-dominate any node that strictly post-dominates m.*

Definition 5. ***Control Dependence:*** *In accordance to [3], node n is* control dependent *on node m in program s if:*

1. *There exists a non-trivial*[3] *path* π *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.*

Example 1 *: Fig. 1 depicts the post-domination concept as well as the control dependency as follows:*

1. s_7 *post-dominates* c_1.
2. s_6 *is the immediate post-dominator of* c_1.

2.4 Strongly Live Variable Dataflow Analysis

Strongly Live Variable (SLV) dataflow analysis computes for each program point, the set of variables whose values in this program point might influence the values on other SLVs. Thus, some SLVs have to be injected at some program points to analyze the program. Based on that, in injecting the slicing criterion as an SLV, the SLV analysis could be employed to compute the data dependencies in the slice. The SLV analysis was used in the PCB-based slicing approach in our previous works [1,2].
The following equations are used in this analysis:

$$\begin{aligned} &S_{entry}(n) = (S_{exit}(n) \setminus kill(n)) \cup gen(n), \\ &\qquad\qquad\qquad\qquad \text{if } kill(n) \subseteq S_{exit}(n) \\ &S_{entry}(n) = S_{exit}(n), \qquad\quad \text{otherwise} \end{aligned} \tag{1}$$

[3] Path π is non-trivial if it contains at least two nodes [7]

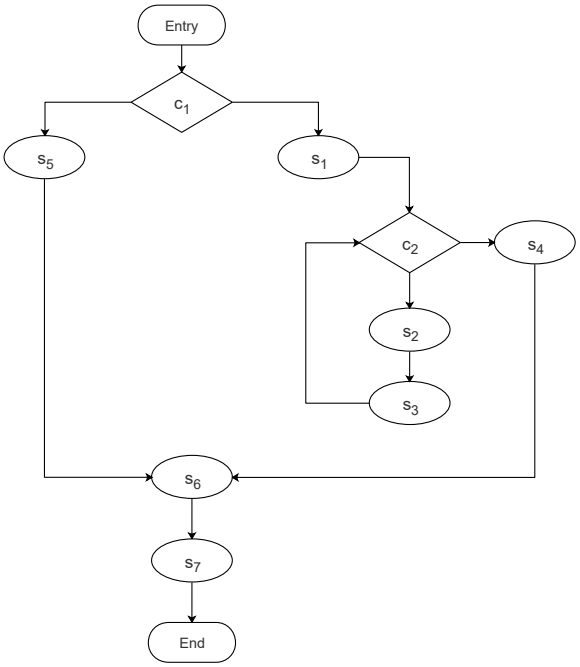

Fig. 1: a Simple CFG.

where, $S_{entry}(n)$ and $S_{exit}(n)$ represent the SLV set before and after the CFG node n, respectively. In the original formulation of SLV analysis [6], the *kill* and *gen* functions are defined for assignments $x := a$, conditions c, and *skip* statements as follows:

$$\begin{aligned} kill(x := a) &= \{x\} \qquad gen(x := a) = FV(a) \\ kill(c) &= \emptyset \\ kill(skip) &= \emptyset \qquad\quad gen(skip) = \emptyset \end{aligned} \tag{2}$$

here, $FV(a)$ denotes the set of program variables that appear in the expression a.

The SLV dataflow analysis can be used to find the data dependencies in static backward slicing. In supposing that an SLV *var* is generated from s', and killed in s, this means - in accordance to SLV questions - that s' is data dependent on s because s defines *var* and s' uses *var*. Hence, SLV dataflow analysis could be exploited to find the definitions that might affect a particular variable in a statement.

3 Program Flows

Structured programs make extensive use of structured constructs such as `if` and `while` conditional statements. In these programs, there is no use of unstructured jumps (e.g. goto in the C language). Conditional statements comprise a predicate and a body, wherein each predicate evaluates a boolean value which determines whether the body is executed or not. Each predicate comprises two successors, wherein one of the successors is executed thrm:thrm3ly if the predicate is true and the other is executed immediately if the predicate is false. Accordingly, the predicate of the conditional statement has two program flows. The main feature of using such blocks is that the flows of the predicates are neither overlapped nor interleaved[4]. Therefore, there are no flows from the body of a conditional statement to the body of another conditional statement. Thus, the structured organisation makes the writing, debugging and understanding of programs easier and straightforward. Unstructured programs comprise both structured and unstructured flows. Consequently, the program may include spaghetti code, where the computations of control dependencies become increasingly complicated.

3.1 Basic Definitions

This subsection defines the program flow and its denotation. Further, it classifies the program flows into three categories: normal, structured and jump flows.

Definition 6. ***The program flow notation*** *($\rightarrow$) refers to the pair of labels defining a program flow, such as:*

$$\ell \rightarrow \ell'$$

where ℓ is the outgoing side of the program flow (from ℓ to ℓ') and ℓ' is the ingoing side of this flow.

Definition 7. *A* ***Normal Program Flow*** *occurs between two labels $a \rightarrow b$ wherein $b = a + 1$.*

Definition 8. *A* ***Jump (Unstructured) Flow*** *is a program flow wherein the outgoing side is an* unstructured jump statement *(e.g.* `goto`*).*

Definition 9. *A* ***Structured Flow*** *is a program flow wherein* the outgoing side is a structured jump statement *(e.g.* `if` *or* `while`*).*
The structured flows in the conditional statements in the While language are defined as follows:

1. `if-then-else`*. suppose the code: if b^c then S_1 else S_2; S'.*
   The flows of [if b^c then S_1 else S_2] are:
   - *$c \rightarrow$ init(S_2): the structured flow.*
   - *$c \rightarrow$ init(S_1).*

[4] The flows overlapping and interleaving are defined in Section 3.2.

- *final*(S_1) $\rightarrow$ *init*(S'): *a jump flow.*
- *final*(S_2) $\rightarrow$ *init*(S').

2. `if-then`. *suppose the code:* *if* b^c *then* S_1; S'.
   The flows of [*if* b^c *then* S_1] *are:*
   - $c \rightarrow$ *init*(S'): *the structured flow.*
   - $c \rightarrow$ *init*(S_1).
   - *final*(S_1) $\rightarrow$ *init*(S'): *a jump flow.*
3. `while`. *suppose the code:* *while* b^c *then* S_1; S'
   The flows of [*while* b^c *then* S_1] *are:*
   - $c \rightarrow$ *init*(S'): *the structured flow.*
   - *final*(S_1) $\rightarrow$ *c: a jump flow.*

As mentioned in Def. 1, in representing a program by a CFG, the program flows are the edges. For the sake of simplicity, these edges could also be denoted by the notation $\rightarrow$, wherein the outgoing node is at the left and the ingoing node is at the right.

3.2 The Categories of Program-flow Interleaving

Herein, the interleaving of program flows is introduced and classified into two main categories, namely overlapping and intersecting.

Definition 10. ***Overlapping Flows:*** *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. 2 depicts the concept of overlapping and intersecting flows.

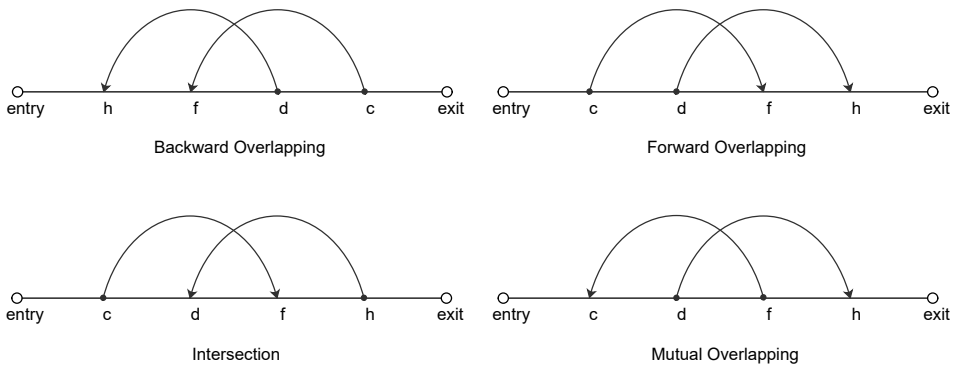

Fig. 2: Overlapping and intersecting flows

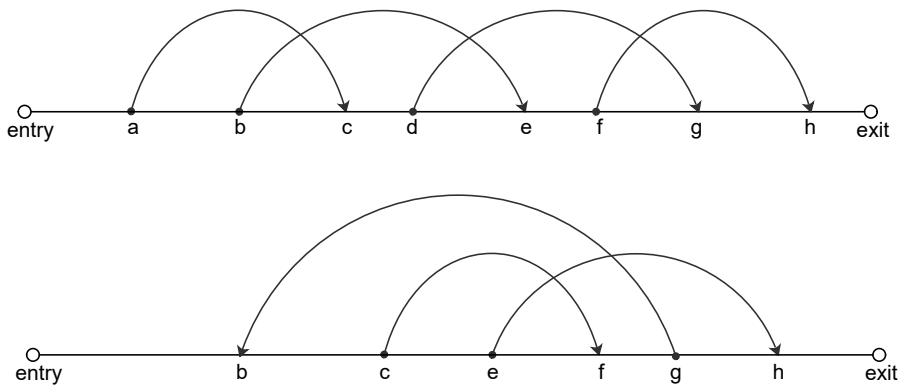

Fig. 3: Two sequences of overlapping flows

Definition 11. ***Sequence of Overlapping Flows*** *(**Overlapping Sequence**) refers to a sequence of flows, where each flow overlaps the previous flow.*

Fig. 3 shows two sequences of overlapping flows.

Definition 12. ***Intersecting Flows*** *refer to the intersection between two program flows ($d \to h$ and $c \to f$) that occurs when $c < d < f$ or $c > d > f$ and $d < c < h$ or $d > c > h$.*

Definition 13. ***The Scope of a predicate*** *lies on the structured flow of p as well as the sequences of overlapping flows where the structured flow of p is involved.*

If we suppose that the structured flow of the predicate p is involved in some sequences of overlapping flows, then the scope of p is the interval from the least ingoing or outgoing side in these sequences to the largest ingoing or outgoing side in these sequences.

Fig. 4 assumes p is a predicate, and its structured flow $p \to t$ is involved in two sequences: $[p \to t, r \to v]$ and $[p \to t, s \to m, o \to k]$. Notice that v is the largest label in these two sequences and k is the least label. Hence, p's scope is from k to v.

Definition 14. ***Bypassing:*** *the program flow $j \to v$* bypasses *the label t if either $j < t < v$ or $j > t \geq v$.*

3.3 Features of Overlapping Jumps

This subsection introduces some facts related to overlapping flows.

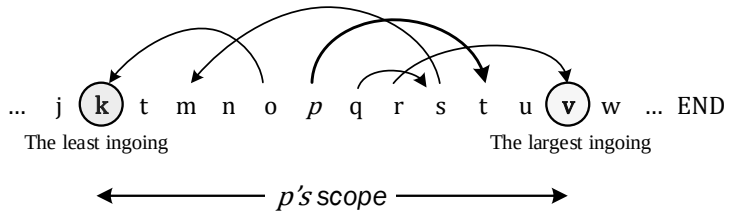

Fig. 4: An example of a predicate scope.

The statement which is not bypassed by any program flow is not controlled by any predicate. When a label i is not bypassed by any program flow, then all the paths from the labels which are less than i to End shall contain i. Thus, i post-dominates all the predicates whose labels are less than i and i is never controlled by such predicates.
On the other side, there is no path from any predicate ahead[5] to i, therefore, and in accordance with Def. 5, no such predicate can control i.

Sequences of overlapping flows can extend the control of predicates. For the conditional statement cs with a predicate c and structured flow $c \to f$, c can control the statements that are not located in the body of cs in accordance to Def. 5, if another unstructured flow overlaps $c \to f$.

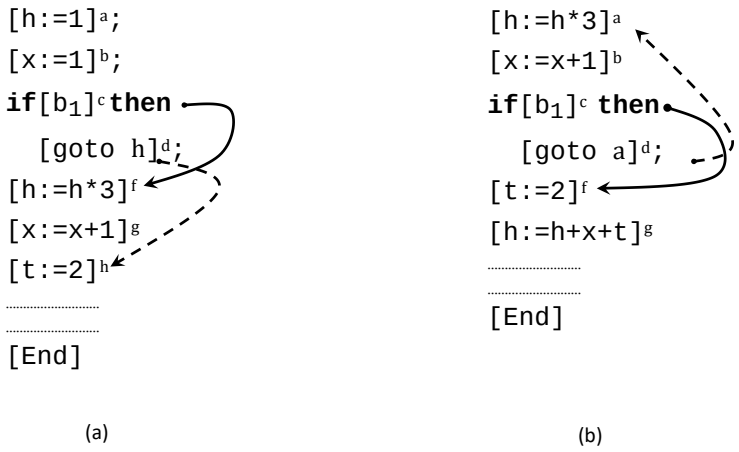

Fig. 5: Extending the power of the predicate by an overlapping flow.

[5] *Predicate ahead* means a predicate with a label larger than i

Fig. 5-a shows how overlapping flows can extend the power of a predicate. In Fig. 5-a, the unstructured flow $d \rightarrow h$ overlaps the structured flow $c \rightarrow f$. As a result, two paths are formed from c: $pth_1 = [c, f, g, ..., end]$ and $pth_2 = [c, d, h, ..., end]$. In investigating the control dependence relation between c and g, we find in pth_1 that all the nodes from c to g are post-dominated by g, which satisfies the first condition in Def. 5. In looking to pth_2, we find that g is not included. So, the second condition in Def. 5 is also satisfied. Consequently, g is control dependent on c. In exploring the paths from c in Fig. 5-b, we can easily conclude that c also controls b in accordance to Def. 5. From the above two examples, we can see that the overlapping flows can extend the power of predicates to control statements outside the boundaries of their bodies.

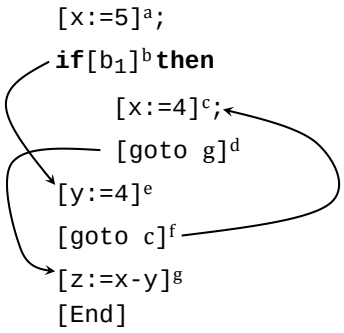

Fig. 6: Shrinking the power of the predicate by a program intersection.

Shrinking the power of predicate by an intersection. In contrast to the overlapping, the intersection between a structured flow and an unstructured flow might shrink the power of a predicate even inside the boundaries of its body. As shown in Fig. 6, the predicate b does not control the statements c and d, because of the intersection between $b \rightarrow e$ and $f \rightarrow c$.

4 On-the-fly Computation of Control Dependencies

This section presents two new theorems that support to compute on the fly and in an accurate manner the predicates that control the possible execution of a statement s under analysis. Practically, the two theorems are applied in two phases. In the first phase, the first theorem is applied to exclude the predicates that are undoubtedly irrelevant. In the second phase, the implementation of the second theorem makes a depth-first search to check potential control dependence relationship between every remaining predicate and s. The theorems rely on the labeling system explained in Section 2.1.

4.1 First Phase: Excluding Irrelevant Predicates

Lemma 1 *Let p be the label of a predicate with scope interval from k to v. Then v post-dominates p.*

Proof.
Since there is no overlapping sequence bypassing v from p, no path can exist from p to End that does not include v. Since all the paths from p to End include v, v post-dominates p □

Lemma 2 *Let p be a label of a predicate such that p is post-dominated by v, 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. 5 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. 5.

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

Lemma 3 *Let p be a label of a predicate with scope interval from k to 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 let p control the execution of w. In accordance with Def. 5-1, 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 unstructured jump flow $\ell \rightarrow \ell'$ [6], where $\ell' \leq i$. We can divide pth into pth^1 and pth^2, where the first is from p to ℓ, and the second is from ℓ' to i. pth^1

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

requires to construct a chain of flows (structured, jump or normal) from p to ℓ. This chain could be formed by placing ℓ in one of the two intervals, in p's scope where $k < \ell < v$ or in v ahead where $v \leq \ell$.

- If $\ell \geq v$: since v is the maximal label in p's scope (Lemma assumption) and v post-dominates p (Lemma 1), all the paths from p to ℓ include v. Accordingly, all the paths from p to j includes v and, based on Lemma 2, j cannot be control dependent on p.
- If $k < \ell < v$: in this case, we get the following facts:
  1. $\ell \rightarrow \ell'$ indeed overlaps one of the overlapping sequences which p is involved with.
  2. To reach j by $\ell \rightarrow \ell'$, ℓ' must be smaller than or equal to j.

  From these two conclusions, the interval of p's scope must start from j and not k. This is contrast to the assumption of the Lemma which states the minimal label in p's scope is k. Therefore, we can state that j could not be control dependent on p if $k < \ell < v$.

Since placing ℓ either in p's scope or ahead of v will not allow p to control the execution of j, the lemma is proved □

In Fig. 4, to establish a forward path from p to j that does not include v, we should create a backward flow f_b, whose outgoing side is in the interval from $[k, v-1]$ and its ingoing side is at j or less than j. In this case, f_b bypasses k and overlaps one of the overlapping sequences including p. As a result, creating the new backward flow f_b will overlap it with one of the overlapping sequences in Fig. 4. Since the ingoing side of f_b is smaller than the smallest label in p's scope, p's scope will be enlarged to include the ingoing side of f_b. Enlarging the p's scope due to establishing a forward path from p to j proves that it is not possible to create f_b if k is the smallest label in p's scope.

Lemma 4 *Let p be a label of a predicate with a scope whose interval ends at v, and w is a label, whose value is larger than v. Then w can not be control dependent on p.*

Proof.
By Lemma 1, v post-dominates p. Therefore, w can not be control dependent on p, because it is in one of the following cases:

1. w does not post-dominate v, so, the first condition in Def. 5 is negated. Hence, it is not possible to make a control dependent relationship between w and p.
2. w post-dominates v. While v post-dominates p, this causes w to post-dominate p as well. This post-domination negates the second condition in Def. 5.

Accordingly, w cannot be control dependent on p. □

In Fig. 4, if we need w to be control dependent on p, then a flow from p's scope to any label larger than w should be established. Since the outgoing side of the proposed flow is in p's scope and its ingoing side is beyond the boundaries of p's scope, it will certainly overlap a flow in p's overlapping sequences. As a consequence, p's scope will be enlarged, and v will be no longer the largest label in p's scope. Based on that, it is not possible to make any control dependence relationship between p and any label larger than v.

Theorem 1. *Let p be a label of a predicate with a scope interval from k to v. Then no possible control dependence relationship can be established between p and a label outside its scope.*

Proof.
Lemma 3 states that it is not possible to establish a control dependence relationship between p and label smaller than k. Lemma 4 states the same thing with labels larger than v. The two lemmas prove that p can not control labels that exist outside the boundaries of its scope. □

4.2 Second Phase: Checking a Potential Control Dependence

Theorem 1 states that in order to create a control dependence relationship between the predicate p and the statement ℓ, ℓ must be inside the interval of p's scope, but not vice versa. This means, some labels in the scope of p might not be control dependent on p. Therefore, we can say that we use the implementation of Theorem 1 to filter out many predicates that surely do not control ℓ, but we must use another technique to check a potential control dependence relationship between the predicate with any label in its scope interval. The first-depth search technique [8], which we call *Exploring Paths by Stopping* or Exploring Paths, can be employed to examine such relationships. This technique conducts a depth-first search in a tree of forward paths whose root is an already determined label ℓ. The search starts at ℓ. Then it moves to visit its immediate successors and so on. Each new visited label is added to a special collection *clctn*. The search does not visit the immediate successors of a special set of labels. These labels are collected in a special list called the *stopping list*. To prevent the occurrence of infinite loops, the search does not add the already visited labels to *clctn*.

Lemma 5 *ℓ' post-dominates ℓ if and only if in Exploring Paths from ℓ (Stopping List = $\{\ell'$,End$\}$), the collection clctn of this search does not contain End.*

Proof.

1. If ℓ' post-dominates ℓ, then ℓ' exists in all the paths from ℓ to End (Def. 2). Since the stopping list contains ℓ', the search that starts from ℓ will indeed cut off at ℓ' and will not continue to End. Consequently, End is never reached and it is never added to *clctn*.
2. If ℓ' does not post-dominate ℓ, then there is a path from ℓ to End that does not contain ℓ' (Def. 2). Hence, Exploring Paths from ℓ will reach End and will add it to *clctn*. □

Lemma 6 *ℓ is control dependent on the predicate with label p, if and only if one of the immediate successors of p is post-dominated by ℓ and the other immediate successor of p is not post-dominated by ℓ.*

Proof.
p has two immediate successors, each of which has a path to End (Def. 1).
If ℓ is control dependent on p, Def. 5 states the followings:

- A path from one of p to ℓ, where ℓ post-dominates every statement in this path except p. Accordingly, one of p's immediate successors is post-dominated by ℓ.
- A path from p to End that does not contain ℓ. As a result, the immediate successor of p in this path must not be post-dominated by ℓ.

As a result, if ℓ is control dependent on p, then ℓ post-dominates one of p's immediate successors, and does not post-dominate the other.
On the other side, if both the following occur: (1) An immediate successor of p is post-dominated by ℓ (2) An immediate successor of p is not post-dominated by ℓ, then ℓ is control dependent on p because (1) satisfies the first condition in Def. 5. and (2) satisfies the second condition in Def. 5.
From the above, the Lemma is proven. □

Theorem 2. *ℓ is control dependent on the predicate p if and only if in Exploring the Paths from the two immediate successors of p, where the stopping list for both explorations is: {ℓ,End}, the collection of one of the explorations will not contain End where the other includes End.*

Proof.
There are two cases as follows:

1. In case that ℓ is control dependent on the predicate p, Lemma 6 states that one of the immediate successors of p (suppose ℓ') is post-dominated by ℓ and the other successor (suppose ℓ'') is not post-dominated by ℓ. Whereas, Lemma 5 states that in exploring the paths from ℓ', the collection does not contain End, while the collection obtained from exploring the paths from ℓ'' shall hold End.
2. In case that ℓ is not control dependent on the predicate p, then in accordance to Lemma 6, either ℓ post-dominates both immediate successors of p or it does not post-dominate any of them. In accordance to Lemma 5, the collections of both explorations either contain or not contain End. □

4.3 Proposed Approach

Definition 15. *$\Pi(\ell)$ is the set of the predicate labels that control the execution of ℓ.*

The predicates of $\Pi(\ell)$ are computed on-demand based on the two phases that are explained in Section 4.1 and 4.2.

First Phase: Collecting Potential Controlling Predicates. This phase implements Theorem 1 to find the predicates that may control the statement ℓ under analysis. The phase finds every program flow i that bypasses ℓ and every sequence of overlapping flows that i is involved in. Afterwards, the flows in these sequences are collected and stored in a set `Flows`[7]. The predicates in these sequences are collected in a set `Predicates`. The procedure is as follows:

1. Add to `Flows` every program flow that bypasses ℓ.
2. Move to 7 if all the items in `Flows` were fetched before.
3. Fetch the flow $i \to m$ from `Flows`.
4. Add to `Flows` every program flow $a \to z$ that overlaps $i \to m$ in accordance to the condition:

$$((i < a < m) \vee (i > a \geq m)) \wedge (z > i \wedge z > m) \vee (z < i \wedge z < m)$$

   if z is a predicate, then add it to `Predicates`.
5. Add to `Flows` every program flow $z \to a$ that is overlapped by $i \to m$ in accordance to the condition:

$$((a < i < z) \vee (a > i \geq z)) \wedge (m > a \wedge m > z) \vee (m < a \wedge m < z)$$

   if z is a predicate, then add it to `Predicates`.
6. Move to 2.
7. Halt the procedure.

Several iterations from 2 to 6 might take place to find all the predicates that control ℓ. This approach is implemented by Algorithm 1.

Second Phase: Exploring Paths by Stopping Technique. This phase implements Theorem 2 to check which items in `Predicates` control the execution of ℓ. The procedure to achieve this is as follows:

1. If `predicates` is empty, then move to 8.
2. Fetch p from `Predicates`.
3. Let ℓ' and ℓ'' be the two immediate successors to p.
4. Let $clct'$ be the collection produced by exploring the paths from ℓ'. The stopping list is {ℓ,End}.
5. Let $clct''$ be the collection produced by exploring the paths from ℓ''. The stopping list is {ℓ,End}.
6. If either $clct'$ or $clct''$ does not contain End and the other collection contains End, then we conclude that ℓ is control dependent on p (Theorem 2).
7. The procedure moves to 1.
8. The procedure halts.

This approach is implemented by Algorithm 2.

[7] The process of storing and retrieving the flows from `Flows` are explained in Algorithm 1.

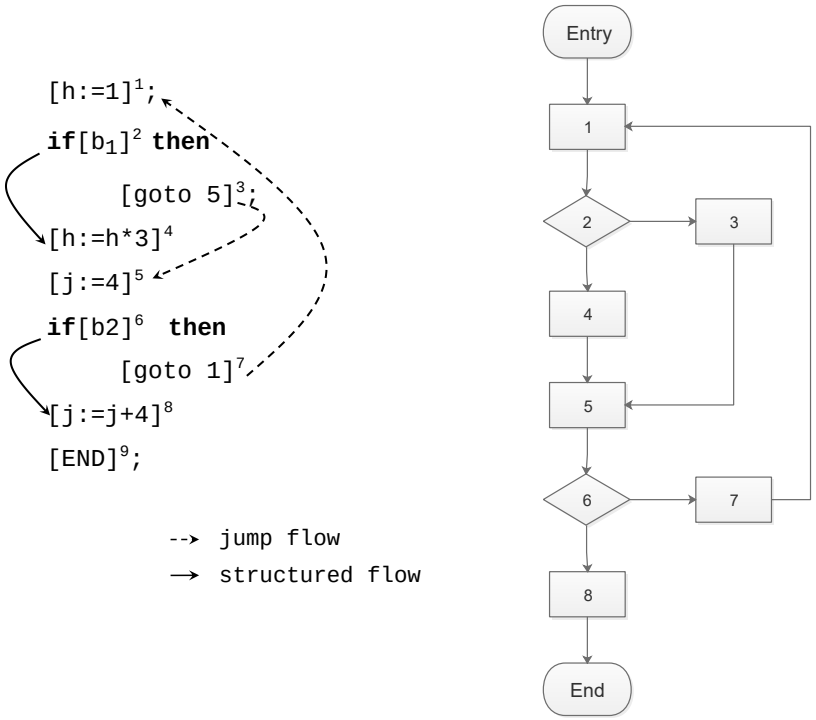

Fig. 7: Unstructured program and its CFG

Example 2 Fig. 7 shows an unstructured program. Hereafter, we will show how the two phases explained in Section 4.3 work together to find the predicates that control the statement labeled by 4.

1. The first phase shows that 4 is bypassed by two overlapping sequences, which are $[6 \rightarrow 8,\ 7 \rightarrow 1]$, and $[2 \rightarrow 4,\ 3 \rightarrow 5]$. The predicates in these sequences are: 2 and 6.

2. In the second phase:
   (a) The second phase for the predicate 2. Since 3 and 4 are the two immediate successors of 2, we have the following collections:
   $clct'(3) = \{3, 5, 6, 7, 8, \text{END}\}$
   $clct''(4) = \{4\}$
   From these two collections, we conclude that 4 is control dependent on 2 (Section 4.3).

(b) The second phase for the predicate 6, for which 7 and 8 are its two immediate successors:
$clct'(7)$= $\{7, 1, 2, 4, 3, 5, 6, 8,END\}$.
$clct''(8)$= {8,END}
Since END exists in both of the two collections, we conclude that 4 is not control dependent on 6.

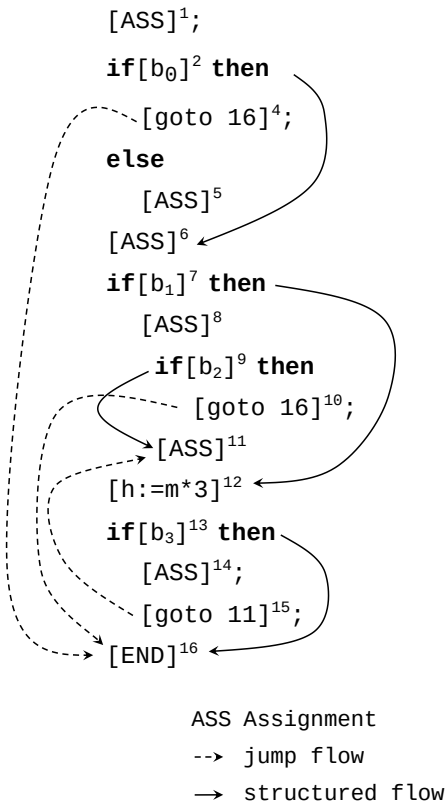

Fig. 8: An example of an unstructured program

Example 3 The phases for finding the predicates that control 12 in Fig. 8 are:

1. In the first phase, 12 is bypassed by the following overlapping sequences: [9 → 11, 10 → END], [7 → 12, 10 → END], [13 → 16, 15 → 11], [2 → 6, 4 → END].
   From these sequences, we find that the set of predicates which might control the execution of 11 are: $\{2, 7, 9, 13\}$.

2. In the second phase, we need to explore the paths from the immediate successor of each predicate computed in the first phase. The stopping list for all the explorations is {12,END}. The explorations are as follows:

   (a) The second phase for 2, whose immediate successors are 4 and 5:
   $clct'(4) = \{4, \text{END}\}$
   $clct''(5) = \{5, 6, 7, 12, 13, \text{END}\}$
   Since the two collections contain END, 11 is not controlled by 2.

   (b) The second phase for 7, whose immediate successors are 8 and 12:
   $clct'(8) = \{8, 9, 10, \text{END}\}$
   $clct''(12) = \{12\}$
   Since END exists in one of the collections and it does not exist in the other collection, 12 is control dependent on 7.

   (c) The second phase for 9, whose immediate successors are 10 and 11:
   $clct'(10) = \{10, \text{END}\}$
   $clct''(11) = \{11, 12\}$
   So, 12 is control dependent on 9.

   (d) The second phase for 13, whose immediate successors are 14 and 16:
   $clct'(14) = \{14, 15, 11, 12\}$
   $clct''(16) = \{\text{END}\}$
   So, 12 is control dependent on 13.

4.4 Proposed Algorithms

The following three algorithms cooperate to compute $\Pi(\ell)$ (Definition 15). Algorithm 1 applies Theorem 1 to find the predicates which might control d. Algorithm 2 applies exploration by stopping to compute the collection from a particular label. Algorithm 3 implements Theorem 2 to compute $\Pi(\ell)$.

Algorithm 1: GetPredicates. Algorithm 1 obtains the complement of the predicates that certainly do not control d. The computations are carried out in conformity with Theorem 1. Algorithm 1 is a direct implementation of the "First Phase" subsection in Section 4.3.

If the `foreach` statement in (4) ranges over all the flows in the program, then this can be a big source of inefficiency. To avoid this inefficiency, the flows are sorted in ascending order with respect to the minimum side in each flow (the minimum of $m \to t$ is $min(m, t) = t$ if $t < m$ and $min(m, t) = m$ if $m < t$). In supposing that we have a program flow $m \to t$ and the label of our statement under analysis is s, there is no use from searching for new flows if the last flow is $m \to t$ and $min(m, t)$ is larger than s. A more intelligent technique[8] can be

[8] A Divide and Conquer algorithm.

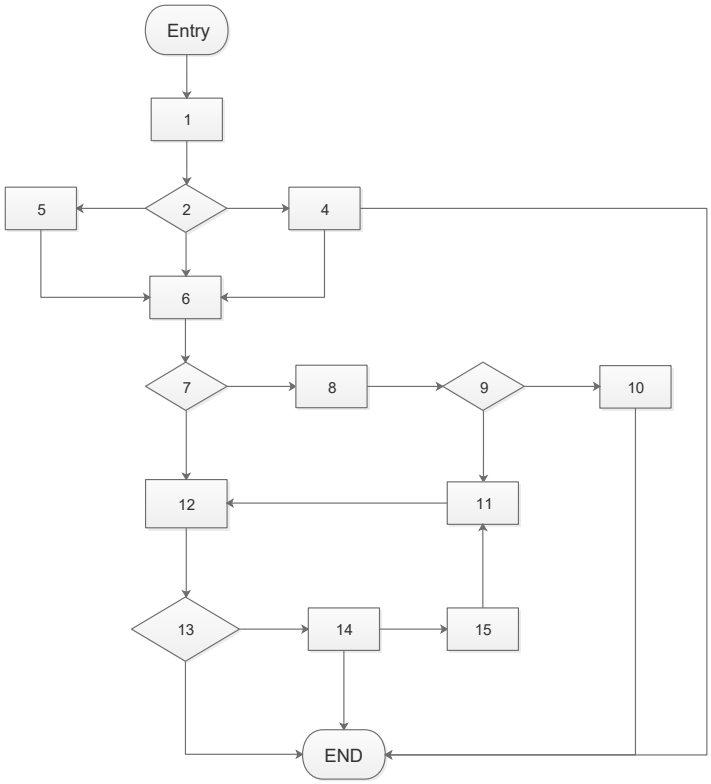

Fig. 9: The CFG of the source code in Fig. 8

applied to avoid starting the search from the beginning. In this technique, we suppose that the flows are stored in an array data structure *Flows* and this array is sorted with respect to the minimum sides of the flows. Let n be the number of the elements in *Flows*, and s is bigger than the minimum side of the flow f which exists at the middle of the array *Flows*. Then we focus on the second half. Otherwise, we focus on the first half. This algorithm is performed recursively until reaching rapidly to the flows bypassing s.

Algorithm 2: ComputeCollection. This algorithm implements the *Exploring Paths by Stopping Technique* from label i with the stopping list $\{\ell, \text{End}\}$. The algorithm has outer and inner loops. The inner loop (7-18) moves from every label in the CFG to its immediate successor(15). If the label has two immediate successors, then it stores one in the *worklist* stack (16-18). The algorithm assigns the current visited label to m. If m is not *clct*, then it is added to *clct* (9). The exploration does not continue to m's successors if m is either in the stopping list (12,13) or in *clct* (11). The outer loop (5-19) uses the *worklist* stack to resume

Algorithm 1: The Implementation of Theorem I

```
1  Procedure GetPredicates(d)
       Input:
       ℓ : a label for which we need to compute the predicates that control it (Π(ℓ))
       Output: Predicates: the predicates that might control the execution of ℓ
       Data: Flows: The flows which are involved in overlapping flows that
             bypass ℓ
2      Predicates := ∅ ;
3      Flows := ∅ ;
4      foreach a → z where ((a < ℓ ∧ z > ℓ) ∨ (a > ℓ ∧ z ≤ ℓ)) do
5          Flows:= Flows ∪ a → z ;
6          if a is a predicate then
7              Predicates:= Predicates ∪ {a}
8      foreach a → z in Flows do
9          foreach i → m where
           ((i < a < m) ∨ (i > a ≥ m)) ∧ (z > i ∧ z > m) ∨ (z < i ∧ z < m) do
10             Flows:= Flows ∪ i → m ;
11             if i is a predicate then
12                 Predicates:= Predicates ∪ {i}
13         foreach i → m where
           ((a < i < z) ∨ (a > i ≥ z)) ∧ (m > a ∧ m > z) ∨ ((m < a ∧ m < z)) do
14             Flows:= Flows ∪ i → m ;
15             if i is a predicate then
16                 Predicates:= Predicates ∪ {i}
17     return Predicates;
```

the searches from the branches which have not been explored (6). The search stops if it reaches *End* (13).

Algorithm 3: DetermineControllingPredicates. This algorithm computes $\Pi(\ell)$. It uses Algorithm 1 to exclude all the predicates that certainly do not control ℓ, and stores the remaining predicates in *Predicates* (2). Then, it assigns every label of *Predicates* to p and obtains the collections of its two successors (4, 5). From these two collections, Algorithm 3 determines whether ℓ is control dependent on p and if it is, the algorithm adds p to $\Pi(\ell)$.

Algorithm 2: The Implementation of Exploring by Stopping Technique

```
1  Procedure ComputeCollection(i, ℓ)
       Input:
       i: The label where exploring the paths search starts
       ℓ: The statement that requires checking its control dependencies
       Output:
       clct: A set of statements and predicates
       Data:
       worklist: a stack of labels.
2      clct = ∅;
3      worklist = ∅;
4      worklist.push(i);
5      repeat
6          m := worklist.pop();
7          while true do
8              if m ∉clct then
9                  clct:=clct ∪ m;
10             else
11                 break;
12             if m = ℓ then break ;
13             if m is End then return clct ;
14             tmp := m ;
               // Fetches the first immediate successor of the label m
15             m:= FirstImmSucc(m);
               // Fetches the second immediate successor of the label m
               // returns NULL if tmp only has one immediate successor.
16             ss := SecondImmSucc(tmp) ;
17             if ss ≠ NULL ∧ ss ∉ worklist then
18                 worklist.push(ss);
19     until size(worklist) = 0;
20     return clct;
```

Algorithm 3: Computing the Control Dependencies in Unstructred Programs

```
1  Procedure DetermineControllingPredicates(ℓ)
       Input:
       ℓ: The statement label that requires finding the predicates controlling it
       Output:
       Π(ℓ): The set of Predicates that control the execution of ℓ
       Data:
       Predicates: a set of predicates
       clct′: The first collection of a predicate
       clct″: The second collection of the same predicate
2      Predicates := GetPredicates(ℓ) ;
3      foreach p ∈ Predicates do
4          clct′:= ComputeCollection(FirstImmSucc(p),ℓ) ;
5          clct″:= ComputeCollection(SecondImmSucc(p),ℓ) ;
6          if (End ∉ clct′ ∧ End ∈ clct″) ∨ (End ∉ clct″ ∧ End ∈ clct′) then
7              Π(ℓ) := Π(ℓ) ∪ p ;
8      return Π(ℓ)
```

5 Optimization

Section 4 proposed a two-phase approach that finds the set of predicates controlling a particular label ℓ, wherein the first stage is approximate but fast, while the second stage has a high overhead but is exact. This section proposes a new phase that is applied before the previous two phases. The main feature of the new phase is that it may determine the control dependencies of ℓ much faster than the previous two phases and save a considerable amount of time. Similar to the demonstration in Section 4, the new phase is formed in a theorem.

The new optimization makes a swift resolution for the statement ℓ under analysis. The resolution is based on a specific attribute of the conditional statement where ℓ stays inside its body. The attribute is realized from comprising ingoing or outgoing sides of unstructured jump flows. If ℓ exists inside many nested conditional statements, then - the new optimization - applies between ℓ and the innermost conditional statement containing ℓ. In this context, when we say that the conditional statement *cs* does not comprise a jump flow, that means neither *cs* nor any of its internal conditional statements has an ingoing or outgoing side of a jump flow.

Definition 16. ***The Conditional Statement of a Label:***
The conditional statement cs of a label i refers to the innermost conditional statement where i exists.

For instance, the predicate of $[\text{ASS}]^{11}$ in Fig. 10 is 10. It is neither 4 nor 6. The predicate of $[\text{ASS}]^{13}$ is 4, and the predicate of $[\text{b2}]^{6}$ is 4.

Lemma 7 *Let c be a predicate label of a conditional statement cs, which does not comprise any jump flow (it has neither an ingoing or outgoing side of a jump flow), and let cs be the conditional statement of the label i. Then, all the paths from other predicates existing outside cs to i contain c.*

Proof.
From the assumptions of the lemma, *cs* does not comprise the ingoing or outgoing side of any jump flow, this makes the flows from c to i the only way to reach i from outside *cs*. Hence, c is included in all the paths that begin outside *cs* and reach i □

Definition 17. *The immediate next statement next(cs) of a conditional statement cs is:*

- *If cs is followed by s (that is, cs;s): next(cs) = init(s).*
- *If cs is the last statement in the program: next(cs) = End.*

Lemma 8 *Let s a non-goto elementary statement, or a conditional statement that does not comprise any jump flow. Then next(s) post-dominates the statement s.*

Proof.
There are two cases for s; a conditional statement or a non-goto elementary statement.
Suppose s is a non-goto elementary statement. Then there is a normal flow (Def. 7) from s to $next(s)$. Since this flow must be included in all the paths from s to End, $next(s)$ post-dominates s. For the second case. Suppose s is a conditional statement. There is path from each statement in s to End (Def. 1). Further, there is a path from $next(s)$ to End (Def. 1). Since there is no jump flow from inside s, the only flows out from s are *final(s)* (Section 2.2), and $next(s)$ is the ingoing side of each of these flows. Therefore, all the paths from the statements in s to End include $next(s)$. In other words, $next(s)$ post-dominates all the statements in s.
The lemma is proved for the two cases and this proves the lemma □

Lemma 9 *Let i be a label with conditional statement cs, and let c be the predicate of cs. Further, let cs not to comprise jump flows. Then there is always a path from c to i, and i post-dominates all the statements in this path except i and c.*

Proof.
Without loss of generality, if cs has two bodies S and S' then we assume that i is in S. The assumption of the lemma states that i is a label with conditional statement cs. That means, in accordance to Def. 16, cs is the innermost conditional statement for i and i could not exist inside an internal conditional statement in S.
The Flow functions (Section 2.2) for the three conditional statements in the While language show that there is always a flow from c to $init(S)$.
We suppose that S has many statements, and S is a composite statement[9] that consists of many elementary and\or inner conditional statements $S = [s_1; s_2; ...; s_n]$. Notice that $init(S) = init(s_1)$, $i = s_k$ where $1 \leq k \leq n$, and $n \geq 1$). Herein, the interesting part in the proof is $[s_1; ..; s_k]$. The definition of the Flow function (Section 2.2) states that there are program flows from each statement s in S to its next immediate statement $next(s)$ if cs does not comprise goto statements (the assumption of the lemma provides this). Since every statement s in S is post-dominated by $next(s)$ (Lemma 8), each statement in S post-dominates its predecessors in S. Based on that, for the path $[c, s_1, ..., s_k]$, s_k post-dominates the statements from s_1 to s_{k-1}, but it does not post-dominate c because c is the outgoing side of two flows (c is a predicate). Hence, the lemma is proved □

Theorem 3. *Let i be a label with conditional statement cs, and let c be the predicate of the conditional statement cs. Further, assume that cs does not comprise any jump flow. Then i is control dependent on c, and it is not control dependent on any other predicate.*

[9] The composite statement in Section 2.2 is formed from two statements and here - for the sake of simplicity - we consider that S is formed from many statements.

Proof.
For proving this theorem we need to prove the following facts: (1) i is control dependent on c, (2) i could not be control dependent on predicates outside cs, and (3) i could not be control dependent on predicates inside cs except c.
Proof of (1)
We have two cases, in the first case cs has one body and in the second case cs has two bodies:

1. In the case that cs has only one body S (e.g. `if` c then S), there are two paths from c:
   (a) The first path starts in the flow $c \rightarrow next(cs)$. This flow bypasses all the statements in S. Since $next(cs)$ post-dominates all the statements in cs (Lemma 5), there is a path from $next(cs)$ to End. As a consequence, there is a path from c to End that does not include i. Hence, the second condition in the definition of the control dependence (Def. 5) is satisfied.
   (b) Lemma 9 states that there is a path from c to i that i post-dominates all its labels except c. Accordingly, the first condition in Def. 5 is satisfied.
2. cs has two bodies S_1 and S_2 (e.g. *if* b^c then S_1 else S_2): if the internal conditional statements in S_1 and S_2 are addressed as the elementary statements which have one predecessor and one successor, then there is one path from c through S_1 to $next(cs)$ and further on to End. On the other side, there is another path from c through S_2 to $next(cs)$ and further on to End. The two paths are as follows:
   (a) In the body which contains i, there is a path from c to i, where i post-dominates each statement in the path except c (Lemma 9). Hence, the first condition in Def. 5 is satisfied.
   (b) There is a path from c to $next(cs)$ through S_2. In accordance to the definition of Flow, this path cannot contain any label in S_1. Therefore, i which is in S_1 cannot be in this path. In taking into account that $next(cs)$ post-dominates all the statements in cs (Lemma 8), the second condition in Def. 5 is satisfied.

Based on 1a, 1b, 2a, and 2b, i is control dependent on c.
Proof of (2)
Let p be a predicate of a conditional statement outside cs. Since any path from p to i must include c (Lemma 7), and i does not post-dominate c, there is no path from p to i where i post-dominate all the statements except p. Based on that, it is not possible to construct a control dependence relationship between i and p because the first condition in Def. 5 is violated.
Proof of (3)
Suppose the body S of cs includes an internal conditional statement i as well as the inner conditional statement cs' whose predicate is c'. Since the structured flow of c' does not bypass i, then c' could not control i (Thrm 1). Hence, c' could not control i and (3) is proved.

From proving (1), (2) and (3), the theorem is proved □

Example 4 Figure 10 shows an example. Assume that the predicates controlling 12 and 6 are demanded.

- For 12: it is control dependent only on 10, because its conditional statement does not comprise any jump flow. Accordingly, there is no need to apply Theorems 1 and 2 for finding the control dependencies that control 12.
- For 6: it is a label inside the conditional statement whose predicate is 4. Since this conditional statement has an ingoing side of unstructured flow at 5, Theorems 1 and 2 must be applied to find the predicates that control 6. The phases for finding the predicates that control 6 are as follows:
  1. **The First Phase**: 12 is bypassed by the following overlapping sequences: [16 $\rightarrow$ 5, 15 $\rightarrow$ END], [2 $\rightarrow$ 15, 1 $\rightarrow$ 3] [4 $\rightarrow$ 13]. From these sequences, we find that the set of predicates which might control the execution of 11 are: {1,4,15}.
  2. **The Second Phase:**,the paths from the immediate successor of each predicate computed in the first phase is explored. The stopping list for all the explorations is {6,END}.

     (a) The Second Phase for 1, whose immediate successors are 2 and 3:
     $clct'(4) = \{2, 15, \text{END}\}$
     $clct''(3) = \{3, 4, 14, 15, \text{END}\}$
     Since the two collections contain END, 6 is not control dependent on 1.

     (b) The Second Phase for 4, whose immediate successors are 5 and 14:
     $clct'(5) = \{5, 6, 13, 14, 15, \text{END}\}$
     $clct''(14) = \{14, 15, \text{END}\}$
     Since the two collections contain END, 6 is not control dependent on 4.

     (c) The Second Phase for 15, whose immediate successors are 16 and End:
     $clct'(\text{End}) = \{\text{END}\}$
     $clct''(16) = \{16, 5, 6\}$
     Since one of the collections contain END, 6 is control dependent on 15.

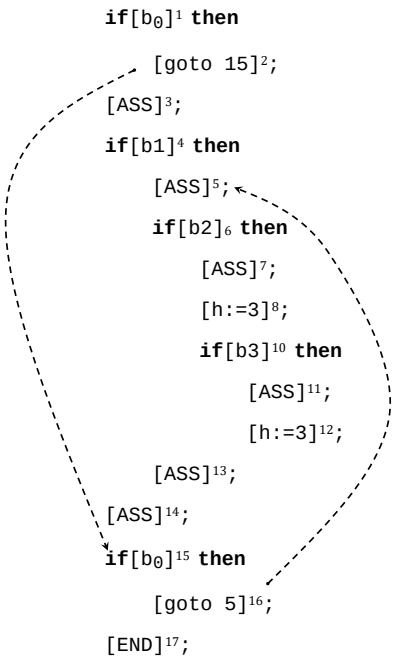

Fig. 10: Example to demonstrate the application of Theorem 3

6 Predicated Code Block (PCB) Graph Representation

Since 1971, the CFG has been acting as the primary and most common program representation. The CFG focuses on modelling the program flows, but it neglects the location information such as the hierarchical structure of the nesting conditional statements, the order of the labels inside the conditional statement, and the original location of each conditional statement. For example, Figure 9 does not illustrate the relation between the labels 7 and 11, although a quick look at the source code in Figure 8 reveals that 11 is inside the body of the predicate 7. This example illustrates a limitation in the CFGs because of neglecting the location information.

To implement Theorem 3 in CFGs, a large amount of annotations should be added to recognize the labels inside each conditional statement as well as the hierarchical structure of the child-parent conditional statements. Rather than adding these annotations to CFGs, a new program representation is presented in our previous work [1,2] for structured programs. In this report, we extend the new representation to include unstructured programs. This extension enables us to use the location information as well as the program structure beside the flow information for finding the control dependencies in the presence of unstructured jumps.

6.1 Constructing the PCBs

Our previous work in [1,2] introduced a new program representation that is referred to as the *PCB graph*. The main unit in this graph is the PCB, which represents a conditional statement. The PCB is mainly formed from a sequence of labels, whose first element points to a predicate of a conditional statement cs. The remaining elements are labels for the elementary statements and placeholders inside the body of cs. The placeholders preserve the original places of the inner conditional statements in cs. Further, the PCB comprises a *type* flag, which is L for PCBs originating from linear conditional statements (e.g. `if`) and C for PCBs originating from circular statements (e.g. `while`). The `if-then-else` statement is translated into two PCBs, one PCB for each branch.

Informally[10], Figure 11 shows how the PCBs could be derived from the conditional statements in the While language. This figure shows a simplified version of the translation where all the statements inside the body of the conditional statement are elementary statements (es). Thus, no placeholders exist.

Following [2], in the PCB graph, the original place of each conditional statement is replaced by a placeholder. A `skip` placeholder supersedes every `if` or `while` statement. The `if-then-else` conditional statement is replaced by an `in-child` placeholder. The PCBs are connected by *interfaces*, and, every PCB is connected with the placeholder of the original conditional statement.
The following points should be taken into account while constructing the PCBs.

[10] A full and formal definition that shows how the PCB blocks are derived and connected from the program syntax is given in Section 6.2

$$\begin{aligned}
\textit{if } c^a \textit{ then } s^\ell; ...; s^{\ell'} &\Rightarrow ([c^a, s^\ell, ..., s^{\ell'}], L)^a \\
\textit{while } c^a \textit{ then } s^\ell; ...; s^{\ell'} &\Rightarrow ([c^a, s^\ell, ..., s^{\ell'}], C)^a \\
\textit{if } c^{a,b} \textit{ then } s^\ell; ...; s^{\ell'} \textit{ else } s^k; ...; s^z &\Rightarrow ([c^a, s^\ell, ..., s^{\ell'}], L)^a, ([\neg c^b, s^k, ..., s^z], L)^b
\end{aligned}$$

Fig. 11: Constructing PCBs from conditional statements and the procedure.

- The PCB inherits the predicate label of the conditional statement.
- The conditional statement itself has a label. When constructing a PCB graph, the placeholder inherits this label.
- Regarding `if-then-else` statements:
  1. Each `if-then-else` predicate has two distinct labels.
  2. A PCB is generated from each branch in the `if-then-else` statement.
  3. Both PCBs are connected to the same `in-child` placeholder.
  4. The predicate of the second PCB is a negation of the `if-then-else` predicate.

Figure 12 shows an example of a program represented by a PCB graph. There are four PCBs, P^0, P^3, P^8, and P^{12}. Three points should be considered in this regard. First, there are two labels for each statement or predicate in Fig 12-b, where the first is a global label. It exists at the right superscript. The second label is a local index. (e.g. the global label 5 corresponds to $P_0[3]$). Second, each PCB has a label (e.g. P_8) and this label is similar to the predicate label. Finally, the placeholder inherits the predicate label (e.g. the statement $[\text{skip}]^{11}$ in Figure 12 inherits the global label 11).

In order to apply the above requirements, the syntax of the While language should be updated to give the predicates for `if-then-else` statements two labels instead of one as follows:

$$\begin{aligned}
cs &::= [\texttt{if } [b]^\ell \texttt{ then } s']^{\ell'} \mid [\texttt{if } [b]^{\ell,\ell'} \texttt{ then } s' \texttt{ else } s'']^{\ell''} \mid [\texttt{while } [b]^\ell \texttt{ do } s']^{\ell'} \\
es &::= [x := a]^\ell \mid [skip]^\ell \mid [goto\ \ell']^\ell \\
s &::= es \mid s'; s'' \mid cs
\end{aligned}$$

6.2 Connecting the PCBs

The second step that comes after constructing the PCBs is to connect them. In a PCB graph, the PCBs are connected through interfaces. Below, the symbol ϵ denotes the set of interfaces in a PCB graph. The interfaces are represented as a pair of labels $(\ell_1, \ell_2) \in \epsilon$, referred to as $\ell_1 \hookrightarrow \ell_2$.
Suppose p' is a child PCB in p, and $p[n]$ is the placeholder of p' in p, then interfaces are created in a PCB graph as follows:

1. The interface $p[n-1] \hookrightarrow p'[0]$ is constructed if $p[n-1]$ is not a `goto` statement. For instance: $P_0[1] \hookrightarrow P_3[0]$ (Fig. 12). If we suppose that the statement $P_0[1]$ is [goto 6][1], then no interface is constructed between $P_0[1]$ and $P_3[0]$.
2. If p' is originated from a `while` conditional statement, then the following interface has to be created: $p'[0] \hookrightarrow p[n]$ (e.g. $P_{12}[0] \hookrightarrow P_0[8]$ in Figure 12).
3. If p' is originated from an `if` conditional statement, and if the last label in p' is not a `goto` statement, then it is connected to p' placeholder (e.g. $P_3[1] \hookrightarrow P_0[2]$ in Fig. 12).
4. The unstructured flow is translated to an interface (e.g.$P_8[1] \hookrightarrow P_0[9]$ in Figure 12).
5. If *cs* is an `if-then-else` conditional statement, *cs* exists inside the PCB p, and is replaced by the placeholder $p[n]$ in the parent PCB p. In this case *cs* is translated into two PCBs (p' and p''), wherein p' represents the first branch, while p'' represents the `else` branch. In accordance to the above assumption, the following interfaces should be created:
   (a) $p'[0] \hookrightarrow p''[0]$.
   (b) $p'[w] \hookrightarrow p[n]$ where $p'[w]$ is the last label in p' and it is not a `goto` statement.
   (c) $p''[z] \hookrightarrow p[n]$ where $p''[z]$ is the last label in p'' and it is not a `goto` statement.

6.3 The Formal Definition for Constructing PCB graphs

Definition 18. *A **PCB graph** is a triple (P, ϕ, ϵ), consisting of a set of PCBs P, a map from global labels to PCBs, ϕ, and a set of interfaces, ϵ, represented as pairs of labels.*

The While program is a single statement s that is a composite of inner elementary and conditional statements $[s_1, s_2, ..., s_n]$. Figure. 13 shows the formal definition that constructs a PCB graph from s. These equations translate each conditional statement *cs* to a PCB graph (P', ϕ', ϵ')[11]. P' is a union of p and P, wherein p is the PCB of *cs*, and P' is the set of PCBs in the PCB graphs of the internal statements in *cs*. The interfaces ϵ' and label maps ϕ' are obtained in a similar way. To compute P, the equations dig deep into the hierarchical structure of the nesting statements until reaching the most inner conditional statements. These statements have only elementary statements. Thus, their PCB graphs are constructed of single PCBs. However, in starting from the innermost conditional statements, the PCB graph for each conditional statement is called to build the PCB graph of the parent conditional statement, and so on until the complete PCB graph of the whole program is built.

After computing the PCB graph for a particular internal statement s', this graph is assigned to $\lambda(s')$, which is called later to form the PCB graph for the parent conditional statement. There are three parameters that are assigned to or obtained from λ_s.

[11] We point to the notations in Figure 13.

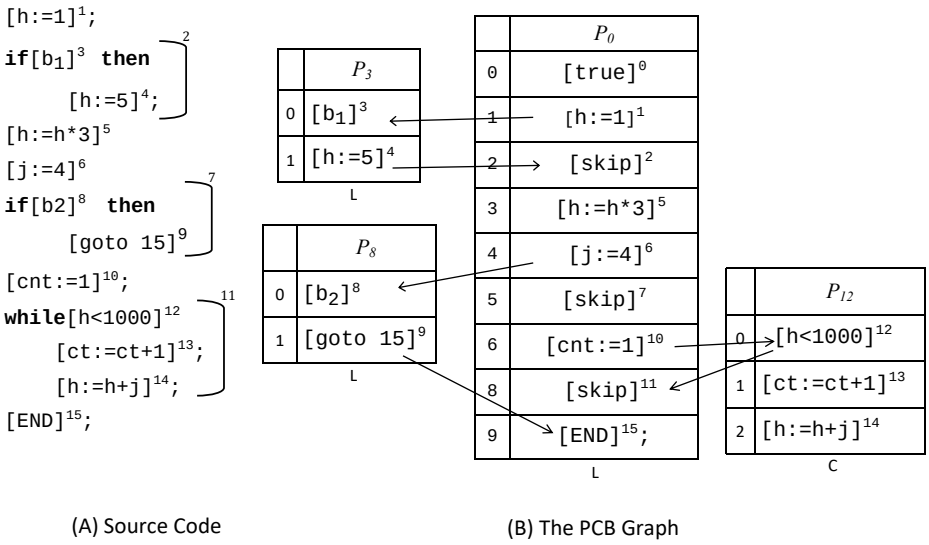

Fig. 12: PCB graph for an unstructured program

$$\lambda_f(s')^\ell = \mathring{s}, (P, \phi, \epsilon), k \tag{3}$$

$\mathring{s}$ determines how s is represented in the parent PCB. If s' is an elementary statement, then $\mathring{s}$ equals s'. Otherwise, if s' is a conditional statement, then $\mathring{s}$ is a placeholder. If s' is a composite statement $s_1; s_2$, then $\mathring{s}$ is a sequence of placeholders and\or elementary statements. k is the last label in $\mathring{s}$. Accordingly, k holds the label of $\mathring{s}$ if $\mathring{s}$ is an elementary statement or a placeholder. Otherwise, k holds the last label in $\mathring{s}$ if $\mathring{s}$ is a composite statement.
As an example, $\lambda_f([\texttt{if } b^\ell \texttt{ then } s]^{\ell'})$ is defined as follows:

$$\lambda_f([\texttt{if } b^\ell \texttt{ then } s]^{\ell'}) = [skip]^{\ell'}, (P', \phi', \epsilon'), \ell' \tag{4}$$

From this equation, we figure out that $[\texttt{if } b^\ell \texttt{ then } s]^{\ell'}$ is replaced by a $[skip]^{\ell'}$ placeholder in its parent PCB. This placeholder inherits its global label ℓ' from the `if` statement. Further, the PCB graph (P', ϕ', ϵ') is computed for this `if` statement and assigned to $\lambda_f([\texttt{if } b^\ell \texttt{ then } s]^{\ell'})$. The equations in Figure 13 show that P' is the outcome of the union of p (the PCB of $[\texttt{if } b^\ell \texttt{ then } s]^{\ell'}$) and P, which is the set of PCBs in the PCB graph representing s. Finally, the last label of $[\texttt{if } b^\ell \texttt{ then } s]^{\ell'}$ in its parent PCB is ℓ'.
For connecting the PCB with other PCBs, the former label to the placeholder of the PCB is often required. In Figure 13, this label is denoted by f, and it is passed to λ by the caller of λ.
The top-level translation of the program starts from:

$$\lambda(s) = (P \cup p,\ \phi[0 \mapsto p],\ \epsilon) \\ \text{, where } es, (P, \phi, \epsilon) = \lambda_0(s), \text{ and } p = \{true^0 : es, L\} \tag{5}$$

In Fig. 13, the symbol $+\!\!+$ stands for concatenation of sequences.
The symbol : stands for the standard cons operator, i.e., $b : [es_1, \cdots] = [b, es_1, \cdots]$.

$$
\begin{aligned}
&\lambda_f([x := a]^\ell) && = [x := a]^\ell, (\emptyset, \emptyset, \emptyset), \ell \\
&\lambda_f([skip]^\ell) && = [skip]^\ell, (\emptyset, \emptyset, \emptyset), \ell \\
&\lambda_f([\texttt{if } b^\ell \texttt{ then } s]^{\ell'}) && = [skip]^{\ell'}, (P', \phi', \epsilon'), \ell' \\
&\quad \textit{where } es, (P, \phi, \epsilon), k = \lambda_\ell(s) \textit{ and } \phi' = \phi[\ell \mapsto p] \\
&\quad \textit{and } p = \{b^\ell : es, L\} \textit{ and } P' = P \cup p \\
&\quad \textit{and } \epsilon_1 = \{f \hookrightarrow \ell\} \quad \textit{if } f \textit{ is not goto,} \\
&\qquad \epsilon_1 = \emptyset \quad \textit{if } f \textit{ is goto} \\
&\quad \textit{and } \epsilon_2 = \{k \hookrightarrow \ell'\} \quad \textit{if } k \textit{ is not goto,} \\
&\qquad \epsilon_2 = \emptyset \quad \textit{if } k \textit{ is goto.} \\
&\quad \textit{and } \epsilon' = \epsilon \cup \epsilon_1 \cup \epsilon_2 \\
&\lambda_f([\texttt{while } b^\ell \texttt{ do } s]^{\ell'}) && = [skip]^{\ell'}, (P', \phi', \epsilon'), \ell' \\
&\quad \textit{where } es, (P, \phi, \epsilon), k = \lambda_\ell(s) \textit{ and } \phi' = \phi[\ell \mapsto p] \\
&\quad \textit{and } p = \{b^\ell : es, C\} \textit{ and } P' = P \cup p \\
&\quad \textit{and } \epsilon_1 = \{f \hookrightarrow \ell\} \quad \textit{if } f \textit{ is not goto,} \\
&\qquad \epsilon_1 = \emptyset \quad \textit{if } f \textit{ is goto} \\
&\quad \textit{and } \epsilon' = \epsilon_1 \cup \epsilon \cup \{\ell \hookrightarrow \ell'\} \\
&\lambda_f([\texttt{if } b^{\ell,\ell'} \texttt{ then } s \texttt{ else } s']^{\ell''}) && = [in\_child]^{\ell''}, (P'', \phi'', \epsilon''), \ell'' \\
&\quad \textit{where } es, (P, \phi, \epsilon), k = \lambda_\ell(s) \textit{ and } es', (P', \phi', \epsilon'), k' = \lambda_{\ell'}(s') \\
&\quad \textit{and } \phi'' = (\phi \cup \phi')[\ell \mapsto p, \ell' \mapsto p'] \\
&\quad \textit{and } p = \{b^\ell : es, L\} \textit{ and } p' = \{\neg b^{\ell'} : es', L\} \\
&\quad \textit{and } \epsilon_1 = \{f \hookrightarrow \ell, f \hookrightarrow \ell'\} \quad \textit{if } f \textit{ is not goto,} \\
&\qquad \epsilon_1 = \emptyset \quad \textit{if } f \textit{ is goto} \\
&\quad \textit{and } \epsilon_2 = \{k \hookrightarrow \ell''\} \quad \textit{if } k \textit{ is not goto,} \\
&\qquad \epsilon_2 = \emptyset \quad \textit{if } k \textit{ is goto} \\
&\quad \textit{and } \epsilon_3 = \{k' \hookrightarrow \ell''\} \quad \textit{if } k' \textit{ is not goto,} \\
&\qquad \epsilon_3 = \emptyset \quad \textit{if } k' \textit{ is goto} \\
&\quad \textit{and } \epsilon' = \epsilon \cup \epsilon_1 \cup \epsilon_2 \cup \epsilon_3 \\
&\lambda_f(s; s') && = es +\!\!+ es', (P \cup P', \phi \cup \phi', \epsilon \cup \epsilon'), k' \\
&\quad \textit{where } es, (P, \phi, \epsilon), k = \lambda_f(s) \textit{ and } es', (P', \phi', \epsilon'), k' = \lambda_k(s') \\
&\lambda_f([goto\ \ell']^\ell) && = [goto\ \ell']^\ell, (\emptyset, \emptyset, \epsilon') \\
&\quad \textit{where } \epsilon' = \{\ell \hookrightarrow \ell'\}
\end{aligned}
$$

Fig. 13: Computation of PCB graph for unstructured programs.

7 Demand-Driven PCB-Based Slicing for Unstructured Programs

The main feature of the PCB-based slicing approach is in computing the program dependencies concurrently with the program slicing rather than computing the dependencies in prior. The approach in [1,2] works well with inter-procedural well-structured programs. This section aims to make non-executable slices for intra-procedural unstructured programs. The difference in computing the slices for structured and unstructured programs is in considering the presence of arbitrary control flows. Such flows make the computation of the control dependencies more complex. The computation of the control dependencies in unstructured programs is based on Theorems 1, 2, and 3.

There are two reasons to work with PCB graphs instead of CFGs. First, as mentioned before, the PCB graph collects the flow information, location information and the syntactic structure in one graph. On the other hand, CFGs lack location information, which is essential to apply Theorem 3. Second, there is no fully demand-driven slicing approach that is based on the CFG, whereas there is a demand-driven slicing approach based on the PCB graph. This paper aims at extending the existing approach to particularly include unstructured cases.

7.1 PCB-based Slicing Approach

This subsection describes the original PCB-based algorithm that works with only structured flows. This analysis starts by translating the slicing criterion to SLV queries. Afterward, these queries are propagated backward, which causes to *kill* and *generate* them by the SLV functions *kill* and *gen*, respectively (these functions (for PCB blocks) are defined in Fig. 14). For each statement s in the program, *gen(s)* adds SLVs from s, while $kill(s)$ removes some SLVs at s.

Each PCB points to its parent PCB. This encodes a parent-child hierarchy of the conditional statements in the program. Later, this hierarchy is exploited to capture the control dependencies in structured programs.

Each PCB P is associated with a special set S_P to store its dataflow queries. Computation of the slice starts from the slicing criterion, which is a set of pairs $< \ell, v >$, where ℓ is a global label and v is a variable. Initially, the slicing criterion is converted to a local PCB index in order to add it to the single set of the PCBs, which contains ℓ. The dataflow queries are used mainly to compute the data dependencies. The dataflow queries are called SLV queries.

Suppose S_P contains $< i, v >$. When $< i, v >$ is fetched, $< i, v >$ visits the labels from $i - 1$ to 0 if P is linear and, in addition, it visits the labels from the last label in P to $i + 1$ if P is circular. Let e stands for the last statement that $< i, v >$ should visit in P. Hence, $e = 0$ if P is linear and $e = i + 1$ otherwise. If the current visited statement is denoted by $P[j]$, visiting $P[j]$ by $< i, v >$ causes one of the following three cases:

case 1: if $v \notin kill(P[j])$ and $j \neq e$, then S_P remains as is.

$$
\begin{aligned}
kill([x := a]^\ell) &= \{x\} \\
gen([x := a]^\ell) &= FV(a) && \textit{where } FV(a) \textit{ is the set of variables appearing in } a \\
kill([b]^\ell) &= \emptyset && \textit{where } b \textit{ is a boolean expressoin} \\
gen([b]^\ell) &= FV(b) && \textit{where } b \textit{ is a boolean expressoin} \\
& && \textit{and } FV(b) \textit{ is the set of variables appearing in } b \\
kill([goto]^\ell) &= \emptyset \\
gen([goto]^\ell) &= \emptyset \\
kill([in - child]^\ell) &= \{x | x \text{ is a program variable}\} \\
gen([in - child]^\ell) &= \emptyset \\
kill([skip]^\ell) &= \emptyset \\
gen([skip]^\ell) &= \emptyset
\end{aligned}
$$

Fig. 14: kill and gen functions of SLV analysis

case 2: if $v \notin kill(P[j])$ then $< i, v >$ is removed from S_P and v does not proceed any further. If $P[j]$ was not sliced before, then the variables used in $P[j]$ will be generated as SLV queries and added to S_P. As well, $P[j]$ is sliced.

case 3: if $v \notin kill(P[j])$ and $j = e$, then $< i, v >$ is removed from S_P and does not proceed further.

Suppose the PCBs P and P' are connected through the interface $P'[j'] \hookrightarrow P[j]$. When the SLV query $< i, v >$ visits $P[j]$ and is not being killed, $< i, v >$ is reproduced in P' as $< j', v >$.

There are two types of placeholders; *skip* and *in-child*. A *skip* placeholder replaces the original place of every `if` or `while` conditional statement. The *in-child* replaces the original place of every *if-else* statement. The main difference is that *skip* does not kill any visiting SLV, while *in-child* kills them. The reason for the different treatment of SLV queries is that for `if` and `while` there is a program flow that bypasses the body of the conditional statement, this does not happen in the case of if-then-else.

In order to obtain control dependencies for a sliced statement, the predicate s_0 in its PCB has to be sliced if it was not sliced before. This routine has to be recursively applied to the parent predicate until the outermost PCB is reached.

Suppose P is a child of the PCB P'. As soon as $P[k]$ is sliced, $P[0]$ should be sliced if it is not already sliced and the variables used in $P[0]$ are generated as SLV queries. In addition, $P'[0]$ is sliced if it is not sliced before and so on.

The algorithms in Section 7.2 to 7.5 provide the extended version of the PCB-based slicing algorithm that handles unstructured programs. Algorithms from 4 to 7 are similar to those in our previous work [1,2]. The contribution here is in adding Algorithm 8.

7.2 Algorithm 4: Slice

The procedure SLICE calls the SLICEPCB procedure for each PCB whose S_P holds some SLV queries. The procedure does not stop until all the single-sets have become empty. This algorithm is developed to do a new search over all the single-sets if one of the single-sets is not empty.

Algorithm 4: Processing the SLV Queries in the Single Set of a PCB

```
1 Procedure SLICE(PCB, I))
    Data:
    N_slc: sliced labels ;
    Bool : boolean value ;
2   N_slc := ∅ ;
3   Bool := true ;
4   while Bool do
5     Bool := false ;
6     foreach P ∈ PCB do
7       if S_p of P ≠ ∅ then
8         N_slc = SLICEPCB(P, I, N_slc) ;
9         Bool := true ;
    Result: N_slc
```

7.3 Algorithm 5: SlicePCB

The procedure SLICEPCB is the main procedure in this report that slices a PCB P with respect to the SLV queries stored in its single set S_P. In SLICEPCB, there are outer and inner loops. The outer loop fetches individually each SLV query from S_P and removes it from S_P (3), and the inner loop moves each of those SLV queries in a specific determined internal path in P.

The SLVs are fetched individually from S_P (3) until it becomes empty. The last fetched SLV query $< i, v >$ visits the local statements from i to e. The variable e refers to the last statement that can be visited by $< i, v >$, and it is calculated at (5-6)[12]. The variable j refers to the index of the current visited statement. j is calculated from the type of P and the current value of j (8-11). *final(P)* equals to 0 in acyclic (linear) PCBs, and equals n in cyclic PCBs, where n is the largest index in P.

Whenever $< i, v >$ visits $P[j]$, and $P[j]$ does not define v (12), v is reproduced through the interfaces (16), which $P[j]$ is the outgoing side. Otherwise, if $P[j]$ defines v (14), $P[j]$ is sliced (16), the predicates controlling the execution of $P[j]$ are determined and sliced (17), and $< i, v >$ is killed (21, 18). Furthermore, the variables used in $P[j]$ are generated as SLV queries in S_P (18), and reproduced through the interfaces (20).

Given that the SLV query $< i, v >$ visits $P[j]$, the transfer function $f^{j,e}_{<i,v>}(S_P)$

[12] e is calculated from the type of the PCB; Circular (C) or Linear (L). The difference is explained in Section 6.1.

updates the single set S_P based on three cases: (1) if $P[j]$ does not define v and $P[j]$ is that statement which $< i, v >$ can visit, or $P[j]$ defines v, (2) If $P[j]$ is not the last statement that $< i, v >$ can reach in P and $P[j]$ does not define v, and (3) if $P[j]$ defines v and $P[j]$ was not sliced before.

Algorithm 5: Processing the SLV Queries in the Single Set of a PCB

```
1  Procedure SlicePCB(P, I, N_slc))
       Input:
       P: current PCB.
       I: Interfaces
       N_slc: sliced labels
2      while S_P ≠ ∅ do
           // Fetch each SLV query from S_P and remove it from S_P
3          < i, v >:= Select(S_P) ;
4          j := −1;
5          if (P is C and i ≠ final(P)) then e := i + 1 ;
6          else e := 0 ;
7          repeat
8              switch j do
9                  case j = −1 : j := i; break;
10                 case j > 0 : j := j − 1; break;
11                 case j = 0 : j := final(P) ;
12             if (v ∉ kill(P[j])) then
                   // Reproducing v through the interfaces
13                 ReproduceByInterfaces(P[j], N_slc, v, I)
14             else
15                 if (P[j] ∉ N_slc) then
16                     N_slc := N_slc ∪ {P[j]};
17                     N_slc =CntrlDep(P[j], N_slc, I);
18                     S_P := f^{j,e}_{<i,v>}(S_P) ;
19                     foreach x ∈ gen(P[j]) do
                           // Reproducing x through the interfaces
20                         ReproduceByInterfaces(P[j], N_slc, x, I);
21                 break; // Fetch new SLV
22         until j = e;
23     return N_slc;
```

7.4 Algorithm 7: ReproduceByInterfaces

The algorithm ReproduceByInterfaces$(P[j], N_{slc}, v, \mathcal{I})$ reproduces the variable v in the outgoing sides (denoted by $P^{\bullet}[z]$) of the interfaces whose ingoing sides are $P[j]$.

Algorithm 6: The Transfer Function of the PCB P

$f^{j,e}_{<i,v>}(S_P) :=$

$$\begin{cases} S_P \backslash \{< i, v >\} & \text{if } (j = e \wedge v \notin kill(P[j]) \ \vee \\ & \quad (v \in kill(P[j] \wedge P[j] \in N_{slc}) \\ S_P & \text{if } j \neq e \wedge v \notin kill(P[j]) \\ S_P \backslash \{< i, v >\} \cup \{(j-1, u) | u \in gen(P[j])\} & \text{if } v \in kill(P[j]) \wedge P[j] \notin N_{slc} \end{cases}$$

Algorithm 7: Reproducing an SLV Query through an Interface

```
1 Procedure ReproduceByInterfaces(P[j], N_slc, v, I)
2 |  foreach (P•[z] ↪ P[j] ∈ I) do
3 |  |  S_P• := S_P• ∪ {< z, v >};
4 |  return;
```

7.5 Algorithm 8: CntrlDep

The CntrlDep procedure proposes a hybrid technique, which sometimes captures the control dependencies from the hierarchical child-parent structure (4-9), and at other times, by using Algorithm 3 (10-17) that determines which of the two modes suits the label of the statement under analysis. Therefore, we use the function G to indicate whether the PCB P comprises a jump program flow. If yes, it is **true**. Otherwise, it is **false**. Whether a program flow is a jump program flow or not is easily found from the syntax of the program. In the second case ($G(P)$ is **false**) if $j \neq 0$ (this case is supported by Theorem 3), then:

1. $P[0]$ is sliced (9).
2. The variables used by $P[0]$ are generated as SLV queries and stored in $S(P)$ (6).
3. The variables used by $P[0]$ are reproduced through the interfaces (8).
4. The predicates controlling $P[0]$ are obtained and sliced (9).

if $j = 0$, then the same steps in the above are performed for the *parent(P)* predicate.

Algorithm 8: Slicing the Predicates Controlling a Statement

```
1  Procedure CntrlDep(P[j], N_slc, I)
       Input: P[j]: The label for which we need to find the predicates that
              control it.
       N_slc: The set of sliced labels.
       I: The set of interfaces.
       Output: N_slc: A new set of sliced labels.
       Data: Predicates: the set of predicates that control P[j] and be computed
             based on Theorems 1 and 2
       // The First Mode: Finding the Control Dependencies from
          hierarchical child-parent structure
2      k := j ; P' := P ;
3      if j = 0 then P' = Parent(P) ;
4      if ¬G(P') then
5          if (P'[0] ∈ N_slc) then return;
6          S_P := S_P ∪ {(0, v)|v ∈ gen(P'[0])} ;
7          foreach v ∈ gen(P'[0]) do
               // Reproducing v through the interfaces
8              ReproduceByInterfaces(P'[0], N_slc, v, I);
9          N_slc =CntrlDep(P'[0], N_slc, I) ;
       // The Second Mode. The Theorems I and II are applied
10     else
11         Predicates := DetermineControllingPredicates( P[j] );
12         foreach P'[k] ∈ predicates do
13             if (P'[k] ∉ N_slc) then
14                 S_P' := S_P' ∪ {< k, v > |v ∈ gen(P'[k])} ;
15                 foreach v ∈ gen(P'[k]) do
16                     ReproduceByInterfaces(P'[k], N_slc, v, I);
17                 N_slc =CntrlDep(P'[k], N_slc, I) ;
18 return N_slc;
```

Given that *G(P)* is true (10), the predicates controlling $P[j]$ should be determined based on Theorems 1 and 2 by the procedure DetermineControllingPredicates[13] (11). Those predicates are stored in *Predicates* set. Afterwards, each element $P'[k]$ in *Predicates* is sliced (17), and its used variables are generated as SLV queries (14) and reproduced through the interfaces (16). As well, $\Pi(P'[k])$ is computed (17).

[13] Since all the flows in the CFG are represented in the corresponding PCB graph and the global labels in the CFG remains in the corresponding PCB graph, we consider that this procedure works with both PCB graphs and CFGs. We think there is no need to rewrite a special version of this procedure for the PCB graph.

8 Related Work

Weiser [9] proposed the backward slicing for debugging purposes. Weiser's approach is based on dataflow equations. It suffers from being over-approximated with inter-procedural cases and it does not include proper relevant `goto` statements in unstructured programs. Ottenstein and Ottenstein [3] invented the Program Dependence Graph (PDG) for finding slices. In such graphs, the program statements, expressions, inputs, parameters, and global variables are represented by vertexes, while edges represent control and data dependencies. The PDG-based slicing is performed by tracking the dependence edges in the PDG that lead to the vertex under analysis (slicing criterion). This way of using the graph to solve the slicing problem is referred to as reachability analysis. The PDG-based slicing is designed to manipulate intra-procedural and well-structured programs, whereas it could not address inter-procedural and unstructured programs.

Horwitz et al. [10] extended the PDG to System Dependence Graph (SDG), which can represent and slice inter-procedural programs. Thus, the SDG contains special types of vertexes and edges that are used to connect the procedures. Similar to the PDG-based slicing, the new approach deals with the slicing as a reachability problem. Afterwards, the SDG-based slicing was studied and improved by various works. Some works focused on improving the performance of the SDG-based slicing [11,12,13,14]. Binkley [15] studied how to produce executable slices. In another work, Binkley [16] presented a method for slicing programs containing aliasing parameters. Livadas, Liang and others [12,17,18] focused on how to address the passing of the pointers as parameters. Many works balanced the trade-off between the context sensitivity and the accuracy [19,20,21,22]. Sinha et al. [23] suggested a method for finding the arbitrary control dependencies which occur as a result of not returning the procedure to its caller.

In the state-of-the-art slicing techniques, which are based on PDG and SDG, the control dependencies are obtained and concluded from the post-dominator tree [3]. Such trees, which organise the post-domination or domination information, are computed by many algorithms [24,25,26,27,28,29,30,31]. All these works require comprehensive analysis along all the nodes in the CFG. They differ in the time complexity. Lengauer-Tarjan's [30] algorithm is the best known and most widely used algorithm for computing the dominance. Cooper et al. [4] showed a method for computing the control dependencies, which is 2.5 times faster than Lengauer-Tarjan algorithm on real programs, although the worst-case time complexity of the Lengauer-Tarjan algorithm is better than Cooper's method [4].

One of the main challenges that the scientists faced in program slicing for about one decade is in moving the control dependencies from the original program to the slice in unstructured programs. As known, the control dependence relationship is formed due to the existence of two paths, one passing the statement under analysis and the second bypassing it. Sometimes, one of the paths contains one or more `goto` statements, and in this case, these statements have to be included in the slice. Weiser [9] could not properly include relevant goto statements, while Ottenstein [3] did not design the PDG-based slicing to address

unstructured programs. This challenge was addressed in many works [32,33,34]. Ball and Horwitz [34] as well as Choi and Ferrentai [33] proposed Augmented CFG (ACFG) and Augmented PDG (APDG), where the `goto` statements are treated as predicates that have two successors. These works suffer from over-approximation due to including irrelevant or fake predicates [35]. Harman and Danicic extended Agrawal's algorithm and made better results by using a refined criterion for slicing the `goto` statements. This algorithm is imprecise with `switch` statements [23]. Sinha et al. [23] discussed inter-procedural control dependencies which prevent the procedures from returning to their call sites (e.g. `halt`).

Demand-driven slicing approaches were presented by Kraft [36], Sandberg et al. [37], Lisper et al. [38], Atkinson and Griswold [39,40,41,42], and Hajnal and Forgács [43]. The main aim of these works is to find a way for avoiding the unnecessary computations, but none of these works proposed a way for obtaining on the fly the control dependencies in unstructured programs.

The most important feature of the contributions in this paper is that it does not require to compute the post-domination information for all the nodes in order to find a tiny subset of control dependencies in the entire program. Avoiding the use of iterative techniques is the main purpose. The time complexities of such techniques are always linked with the number of nodes or statements in the CFG, PDG or SDG.

9 Discussion and Future Work

This paper presents the first approach that can obtain the control dependencies in unstructured program without making a comprehensive analysis. This approach determines for a particular statement the set of predicates that control its execution. Therefore, there is reason to believe that the new approach is faster than the classical approaches in finding the control dependencies for few number of statements. The successful key of this approach is in using the *syntactic structure* and the *location-based information* (Theorems 1 and 3) besides the *control-flow information* (Theorem 2). These two types of data are collected in one graph by extending the PCB graph to include unstructured cases in unstructured programs. As a result, we get a more efficient program representation for unstructured programs. Finally, the paper presents a new two-mode slicing method which can get the control dependencies from the child-parent hierarchical structure in the PCB graph or from the implementation of Theorem 1 and Theorem 2. This two-mode method introduces a novel way to analyze unstructured codes.

The work in this paper could be extended to slice inter-procedural unstructured programs. Further, it could be improved to reuse the control dependence information between the program statements rather than repeating the same computation for the statements that are controlled by the same predicates. In moving a control dependence relationship from the original program to a slice, it is important to move the two paths causing the emerging of this dependence. In unstructured programs, such paths might include `goto` statements and such statements should be sliced also. Another potential work is to study how to find and include the relevant `goto` statements. Finally, there is a need to make an experimental evaluation to compare the performance of this approach with the state-of-the-art approaches.

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. 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.
4. Keith D Cooper, Timothy J Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Software Practice & Experience*, 4(1-10):1–8, 2001.
5. Rajiv Gupta. Generalized dominators and post-dominators. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 246–257. ACM, 1992.
6. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Berlin Heidelberg, 2015.
7. 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.
8. Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.
9. Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.
10. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990.
11. Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependences types. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 158. IEEE Computer Society, 2001.
12. Panos E Livadas and Stephen Croll. System dependence graph construction for recursive programs. In *Computer Software and Applications Conference, 1993. COMPSAC 93. Proceedings., Seventeenth Annual International*, pages 414–420. IEEE, 1993.
13. Istvan Forgács and Tibor Gyimóthy. An efficient interprocedural slicing method for large programs. 1996.
14. Panos E Livadas and Stephen Croll. System dependence graphs based on parse trees and their use in software maintenance. *Information Sciences*, 76(3-4):197–232, 1994.
15. David Binkley. Precise executable interprocedural slices. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):31–45, 1993.
16. David Binkley. Slicing in the presence of parameter aliasing. In *Software Engineering Research Forum*, pages 261–268, 1993.
17. Panos E Livadas and Theodore Johnson. An optimal algorithm for the construction of the system dependence graph. *Information Sciences*, 125(1-4):99–131, 2000.
18. Donglin Liang and Mary Jean Harrold. Reuse-driven interprocedural slicing in the presence of pointers and recursions. In *International Conference on Software Maintenance*, page 421. IEEE, 1999.

19. Gagan Agrawal and Liang Guo. Evaluating explicitly context-sensitive program slicing. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 6–12. ACM, 2001.
20. David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*, pages 44–53. IEEE, 2003.
21. Jens Krinke. Evaluating context-sensitive slicing and chopping. In *International Conference on Software Maintenance*, page 0022. IEEE, 2002.
22. Jens Krinke. Context-sensitivity matters, but context does not. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 29–35. IEEE, 2004.
23. 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.
24. 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.
25. 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.
26. Stephen Alstrup, Dov Harel, Peter W Lauridsen, and Mikkel Thorup. Dominators in linear time. *SIAM Journal on Computing*, 28(6):2117–2132, 1999.
27. 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.
28. Matthew S Hecht. *Flow analysis of computer programs*. Elsevier Science Inc., 1977.
29. 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.
30. 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.
31. Paul W Purdom Jr and Edward F Moore. Immediate predominators in a directed graph [h]. *Communications of the ACM*, 15(8):777–778, 1972.
32. Hiralal Agrawal. On slicing programs with jump statements. In *ACM Sigplan Notices*, volume 29, pages 302–312. ACM, 1994.
33. 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.
34. Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *International Workshop on Automated and Algorithmic Debugging*, pages 206–222. Springer, 1993.
35. Mark Harman and Sebastian Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance: Research and Practice*, 10(6):415–441, 1998.
36. Johan Kraft. *Enabling timing analysis of complex embedded software systems*. PhD thesis, Mälardalen University, 2010.
37. 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.

38. 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.
39. 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.
40. Leeann Bent, D Atkinson, and W Griswold. A qualitative study of two whole-program slicers for c. *Technical Report*, 2000.
41. 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.
42. 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.
43. 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.