Syntax-Directed Control Dependence Analysis: Eliminating Graph Overhead

Husni Khanfar

School of Innovation, Design, and Engineering, Mälardalen University,
SE-721 23 Västerås, Sweden
husni.khanfar@mdu.se

Abstract. State-of-practice techniques in static program analysis mainly depend on constructing intermediate data structures, primarily graphs, to facilitate various analyses. However, these structures introduce significant computational overhead in terms of time and space. One key static analysis technique is the computation of control dependencies, which plays a crucial role in optimizing compilers, program slicing, debugging, and parallelization by determining the execution order of statements based on conditional branches. In this paper, we present an improved technique for computing control dependencies without requiring any intermediate graph-based representations. Our approach eliminates unnecessary computations, significantly reducing resource consumption while maintaining accuracy. Experimental evaluations demonstrate that our method not only outperforms classical techniques in terms of execution speed but also exhibits superior scalability and stability when applied to large-scale programs. These findings suggest that our approach is a viable alternative to conventional methods, offering a more efficient solution for static program analysis in modern software engineering.

Keywords: Program Analysis, Control Dependence, Syntax-Directed.

1 Introduction

Control dependencies define the relationship between the predicates of conditional statements and the execution of dependent statements in a program. A statement S is control dependent on a predicate P if the execution of S depends on the outcome of P. This concept is essential in various fields, including optimizing compilers, program slicing, program testing, and software verification.

All state-of-the-art approaches compute control dependencies by constructing three layers of intermediate data structures called graphs, each representing a specific aspect of the source code, and each building on the previous one. First, they represent the source code as an Abstract Syntax Tree (AST), which captures the hierarchical structure of the program. From the AST, a Control Flow Graph (CFG) is generated to represent the program's execution flow and to group consecutive statements into basic blocks as nodes. A basic block is a sequence of consecutive statements with a single entry point and a single exit point. Using the CFG, a post-dominator tree is computed, where a node post-dominates another if every path to the exit node passes through it. Control dependencies are derived from this post-domination information.

MRTC Report, Mälardalen Real-Time Research Centre - Mälardalen University - Sweden
ISRN: MDH-MRTC-354/2025-1-SE
Received in 27-Feb-2025 Accepted in 14-March-2025

The most major approaches for computing control dependencies rely on intermediate data structures, or graphs, but there are some variations. Ottensien et al.[1] and Ferrante et al.[2] compute standard control dependencies but shift to a more graph-theoretic approach. Their method relies on post-dominance relations in the Control Flow Graph (CFG), where control dependencies are derived from dominance frontiers in a Postdominator Tree. This representation improves accuracy but struggles with irreducible flow graphs, requiring graph transformations to recover accurate control dependence information. Ramalingam [3] refines Ferrante's method by focusing on optimizing control dependence computation in irreducible graphs. His technique remains within the standard framework of CFG and Postdominator Trees but introduces a transformation mechanism to improve performance in non-structured control flows. Pingali and Bilardi [4] proposed an Augmented Postdominator Tree (APT) as an alternative representation for the Control Dependence Graph, describing the relationships among three types of control dependencies.

Havlak [5] introduces the Loop Nesting Tree for better dependency extraction and utilizes the Control Flow Graph (CFG) for this purpose. Johnson et al. [6] proposed the Program Structure Tree (PST) as a hierarchical approach to control dependence analysis. Ball and Horwitz [8], as well as Choi and Ferrante [7], introduced the Augmented CFG (ACFG) to handle `goto` statements by treating them as predicates.

The works presented above are some of the main contributions in this field, but many other studies follow the same paradigm, employing different or enhanced types of intermediate data structures. However, some researchers [10,11,12,13] have highlighted the high time and space costs required to construct intermediate data structures for computing data and control dependencies. As an alternative, they proposed methods that attempt to compute control and data dependencies directly from the Abstract Syntax Tree (AST) without relying on the construction of a Control Flow Graph (CFG), Postdominator Tree (PDT), or any other graph-based representation. The first three works [10,11,12] cover both structured and unstructured programs[1] , but they fail to provide accurate results for both types. The most recent work [13] is limited to handling subprograms containing `continue` and `break` statements but does not address `return` or `exit` statements.

Historically, all approaches for computing control dependencies have shared the common goal of identifying control dependencies in both structured and unstructured programs. This research dates back to the early 1980s, when many programming languages heavily relied on `goto` statements, and the risks of "spaghetti code" were not well understood.

Modern programming languages such as Python, Java, Kotlin, Rust, and Go implement control flow constructs similar to those in C and C++—including conditional statements, loops, and control transfer statements like `break`, `continue`, and `return`—but they disallow `goto` statements. In contrast, C and C++ still support goto, though many industry coding standards, such as MISRA-C and CERT C, strongly discourage its use due to its potential to create unstructured and difficult-to-maintain code.

This work introduces the first approach capable of accurately computing control dependencies for structured source code in $O(N)$ time, ensuring each statement is processed only once. Unlike existing methods, it does not require transforming the source code into an intermediate representation, such as an Abstract Syntax Tree (AST) or a Control Flow Graph (CFG). Instead, it directly reads and analyzes the source code, achieving highly efficient control dependency computation.

[1] A structured program is one that follows a clear control flow hierarchy using loops (e.g., `while`, `for`) and conditional statements (e.g., `if-else`) without arbitrary jumps. These programs are typically easier to analyze and optimize. In contrast, an unstructured program contains non-sequential control flows, such as `goto` statements or indirect jumps, making them harder to analyze due to complex execution paths.

In this work, the sections are organized as follows: Section 4 introduces the relevant concepts and presents the necessary definitions for our work. Section 5 establishes the theoretical foundations underlying our approach. Section 7 presents the algorithms and demonstrates their connection to the theoretical framework introduced in the previous section. Section 9 provides an empirical evaluation of our approach, comparing its performance with state-of-the-practice techniques. Section 10 reviews and summarizes closely related work. Finally, Section 11 concludes the paper by summarizing our findings, discussing broader implications, and suggesting directions for future research.

This work introduces the first approach capable of accurately computing control dependencies for structured source code without requiring transforming the source code into an intermediate representation, such as AST or CFG. As a result, getting speed outperformas the classical method that relying on intermedite datastructure. the speedup varying from 2 to 50.

One of the key considerations we took into account in this work is facilitating the reader's understanding by providing numerous examples—more than 25 in total.

This work has been tested on over 3 million input cases, and we plan to scale the evaluation to 20 million to further ensure correctness.

In this work, the sections are organized as follows: Section 2 summarizes the basic methodology of the state-of-the-art approach. Section 3 informally introduces the new approach proposed in this work. Section 4 presents the relevant background concepts, while Section 5 establishes the theoretical foundations. Section 6 introduces the primary goals of the new datasets used in this work, and also presents the event–action relationships. Section 7 describes the algorithms in detail. Section 8 focuses on the `if..else` statement as a special case requiring dedicated handling. Section 9 provides the experimental evaluations. Finally, Section 10 discusses related work, and Section 11 concludes the paper.

2 Foundations and Insights from Prior Work

This section summarizes the theoretical foundations of relevant classical works that utilize graph-based representations and analytical techniques.

2.1 Control Dependencies and Post-Domination Facts

The classical approach computes control dependencies based on post-domination facts. A statement X post-dominates statement S if every path from S to End includes X. For example, in Code 1, `f2()` (Label 10) post-dominates Label 1, whereas Label 10 does not post-dominate Label 1 in Code 2.

Control dependency analysis based on post-domination facts is defined as follows: a predicate m controls the execution of n (denoted as $n \rightarrow m$) if and only if the following two properties hold.

- *Property A*: There exists a path from a successor of m to n, where n post-dominates all labels along this path.
- *Property B*: Another path exists from the second successor of m to End, excluding n.

Note: Property A of control dependencies is referenced in this context as Def. 6-A, while Property B is Def. 6-B.

code 1: `while(b`^{1}`){if(b1`^{2}`)continue`^{3}`;f()`^{4}`;if(b2`^{5}`)break`^{6}`;f1()`^{7}`;if(b3`^{8}`)continue`^{9}`;}`
`f2()`^{10}`;`

2.2 Example 1

If we check whether Predicate 2 in Code 1 controls Labels 4 and 5, we begin with the following observations. The two successors of Label 2 are Labels 3 and 4. The only path from Label 3 is [3,1,10,End], which reaches End without including Labels 4 and 5. Therefore, Def. 6-B is satisfied.

Next, we examine the two paths originating from the second successor of Predicate 2, which is Label 4: [4,5,6,7,8,9,1,10,End] and [4,5,7,8,9,1,10,End]. In both paths, Labels 4 and 5 post-dominate Label 4, satisfying Def. 6-A .

Based on this, Predicate 2 controls Labels 4 and 5.

Label 2 does not control Label 6 because Label 6 does not post-dominate Label 4 or Label 3. This occurs because not all paths originating from Label 4 include Label 6.

code 2: `while(b`^{1}`)`$^{\circ}${`if(b2`^{2}`)`$^{\bullet}${`if(b3`^{3}`)`$^{\triangledown}${`if(b4`^{4}`)return`^{5}`;f1()`^{6}`;`}$^{\triangledown}$`f2()`^{7}`;continue`^{8}`;`}$^{\bullet}$`x=5`^{9}`;`}$^{\circ}$
`f3()`^{10}`;`

2.3 Control Flow Graph and Post-Dominator Tree

A Control Flow Graph (CFG) is a directed graph used in program analysis and compiler design to represent the flow of execution in a program.

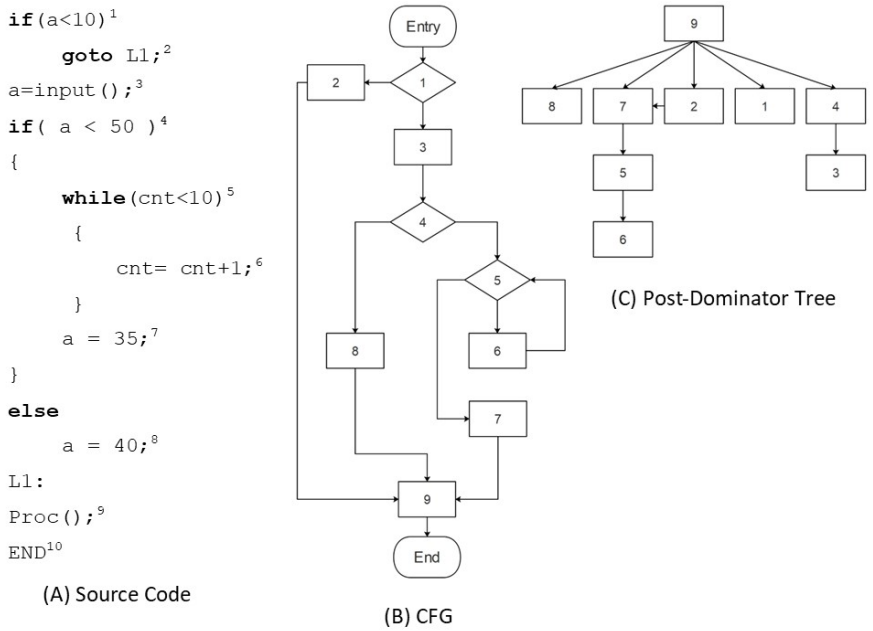

Fig. 1

Definition 1. *The Control Flow Graph (CFG) consists of a set of nodes, which represent statements in intra-procedural programs (single procedure), while the edges denote control flows, as shown in Fig. 1-B.*
The CFG always includes two additional nodes for each represented code: Entry and End.
By definition, a CFG must have a path from Entry to every node in the graph and at least one path from each node to the End node. Otherwise, dead code may exist.

Algorithms that calculate post-domination compute all post-domination information in the program and organize it into a graph or tree. They cannot compute the post-domination fact for a single statement in isolation. Instead, they must first compute and store all post-domination data before using it.

To determine the predicates that control the execution of a specific node in the CFG, all post-domination information in the program must be computed and arranged into a post-dominator tree. Control dependency is then derived from this tree based on the following rule: a node W is control dependent on node U if the CFG contains an edge $U \rightarrow V$, where W post-dominates V but does not post-dominate U.

2.4 Example 2

If we examine whether Node 4 controls Node 7 in Fig. 1-B, we find that Node 4 has two successors: Label 5 and Label 8. From Fig. 1-C, we observe that Label 7 post-dominates Label 5 but does not post-dominate Label 8. Therefore, Label 4 controls Label 7.

2.5 Predicate Code Block (PCB) graph

The approach in this work builds on the Predicated Code Block (PCB) graph, a program representation introduced in [9,14], which designates compound statements[2]as the primary entities in program representation. Each compound statement is represented as a PCB, incorporating the Boolean expression of its predicate and the label of each statement.

Typically, each PCB inherits the label of the Boolean expression of the corresponding conditional statement. For example, in Code 2, P_4=[4,5].

For nested compound statements, a placeholder represents the location of the child PCB within its parent. Similar to a PCB, the placeholder inherits the label of the Boolean expression. For example, P_3=[ph(4),6] and P_2=[ph(3),7,8].

3 Synopsis of the Proposed Approach

This section introduces the proposed approach within this context.

3.1 Predicate-Controlled Regions

In structured source code, a predicate k may control statements in three regions: the first is the follow region of its PCB, the second is the follow region of the loop it resides in, and the third is the header of that loop. It is important to define the *follow region* concisely:

Definition 2. *A loop follow block is the unique basic block (or node) in a control flow graph (CFG) that executes immediately after the termination of a loop. It is the first basic block to be executed once the loop condition evaluates to false and the loop body is no longer executed.*

Definition 3. *A basic block is a sequence of consecutive instructions in a program that executes sequentially from start to finish without interruption, except at the end.*
It begins at a point where control can enter (such as after a label or a branch) and terminates when execution may jump elsewhere (such as a branch, jump, or return statement). Once execution enters a basic block, all instructions within it must execute in order until it reaches the end.

[2] The compound statement is either a conditional statement (e.g., `if`) or a loop (e.g., `while` or `for`).

For example, in Code 3, the follow region of the while conditional statement (P_1) is [11,12,13]. The follow region of P_2 is [4,5,6]. The header region of P_1 consists only of Label 1.

code 3: `while(b1`^{1}`){if(b2`^{2}`)continue`^{3}`; f1()`^{4}`;f2()`^{5}`;if(b3`^{6}`)break`^{7}`;f3()`^{8}`;`
`if(b4`^{9}`)continue`^{10}`;}f4()`^{11}`;f5()`^{12}`;if(b5`^{13}`)return`^{14}`;f6();`

3.2 Identifying Control Dependencies Outside While Loops

In this case, each predicate enclosing a jump statement controls its follow region. Furthermore, it also controls the follow region of its parent PCB if it is the last child PCB within its parent. For example:

In Code 4, the follow region of P_4 is [6,7], meaning 6→4 and 7→4 hold. Similarly, P_7.*follow* = [9,10], resulting in 9→7 and 10→7. Now, P_2.*follow* = [12,13], leading to 12→2 and 13→2. Additionally, the region [12,13] is controlled by Predicate 10.

code 4: `void main(){f1()`^{1}`;if(b1`^{2}`)`•`{f2()`^{3}`;if(b2`^{4}`)return`^{5}`;f3()`^{6}`;if(b3`^{7}`)return`^{8}`;f4()`^{9}`;`
`if(b4`^{10}`)return`^{11}`;`•`}f5()`^{12}`;if(b5`^{13}`)return`^{14}`;f6()`^{15}`;f7();}`

3.3 Identifying Control Dependencies Inside While Loops

Control dependencies inside while loops are recognized in the same way as they are outside while loops. Each predicate that contains a jump statement—`continue`, `break`, or `return`—controls its follow region. Additionally, a predicate also controls the follow region of its parent if it is the last one in its PCB.

In Code 3, `b2` controls Labels 4, 5, and 6, while Predicate 6 controls Labels 8 and 9. Similarly, `b1` controls Labels 11, 12, and 13.

3.4 Control Dependence Analysis of Loop Headers and Follow Blocks

Capturing the control dependencies of loop headers and follow blocks through internal predicates that control or enclose different numbers and types of jump statements is one of the most challenging aspects of this work. This is because they may either support each other in satisfying the two control dependency properties or eliminate each other, depending on their combinations within the PCB.

The main idea of this work is to capture one of the control dependency properties in a query or dataset, then propagate it forward inside the PCB to determine whether the second property is satisfied. The satisfaction of the second property depends on this propagation.

We define five datasets, created within each PCB. Four of these datasets, $\mathcal{E}_h$, $\mathcal{C}_h$, $\mathcal{B}_f$, and $\mathcal{R}_f$, collect each visited predicate that satisfies one of the properties for a specific region:

- $\mathcal{E}_h$ collects predicates that control `break` or `return` statements because they satisfy Def. 6-B for controlling the loop header.
- $\mathcal{C}_h$ collects all predicates that control `continue` statements because they satisfy Def. 6-A in their relation to the loop header.
- $\mathcal{B}_f$ collects all predicates that control `break` statements because they satisfy Def. 6-A in their relation to the follow region of the loop.
- $\mathcal{R}_f$ collects all predicates that control return statements, meaning these predicates satisfy Def. 6-B concerning the follow region of the loop.

– $\mathcal{C}_f$ is designed to capture control dependencies involving both continue and break statements, which may lead to controlling the follow region when a `return` statement is present. This dataset is slightly more complex than the others and is specifically designed to capture special scenarios.

Based on this, in Code 1, P_1 has the sequence [P_2,4,ph(5),7,ph(8)]. The analysis traverses $ph(2)$, and since Predicate 2 controls a `continue` statement, which jumps directly to Label 1, Predicate 2 is added to $\mathcal{C}_h(P_2)$. After processing P_2, $\mathcal{C}_h(P_1)$=$\mathcal{C}_h(P_2)$={2}. When the analysis advances to Label 4, nothing changes. Upon reaching P_5, it processes it and finds a `break` statement, which jumps outside the `while` loop and then reaches End. This satisfies the second property, meaning all predicates in $\mathcal{C}_h(P_1)$ control the execution of Label 1. After establishing a dependence between 1 and each label in $\mathcal{C}_h(P_1)$, $\mathcal{C}_h(P_1)$ is emptied.

4 Background

4.1 Control Flow Graph, Basic Blocks, Post-Domination, and Control Dependency

This section provides an overview of the Control Flow Graph (CFG), basic blocks, and control dependencies.

A CFG models the control structure of a program by representing its execution paths as a directed graph. The nodes in a CFG correspond to *Basic Blocks (BBs)*, which are sequences of consecutive statements with a single entry and exit point. Directed edges between nodes represent possible control flow transitions in the program.

Definition 4 (Control Flow Graph (CFG)). *A CFG for an intra-procedural program s is a 4-tuple $(N, E, Entry, End)$, where N is the set of nodes, each representing a basic block in s; E is the set of directed edges representing control flow transitions between nodes, i.e., $E \subseteq (N \times N)$; Entry is a unique start node, where $Entry \in N$; End is a unique exit node, where $End \in N$; every node $n \in N$ is reachable from Entry; and every node $n \in N$ has a path leading to End.*

Definition 5. Post-domination: *In a CFG G, a node n* post-dominates *another node y if every path from y to End contains n.*

Definition 6 (Standard Control Dependence). *A node n is* standard control dependent *on a node m in program s (denoted as $n \to m$) if and only if the following two properties hold: (A) there exists a non-trivial[3] path π from m to n, such that every intermediate node $n' \in (\pi - \{m, n\})$ is post-dominated by n, and (B) m is not strictly post-dominated by n.*

4.2 Program Representation: Predicate Code Block Graph

The Predicate Code Block graph (PCB-graph), introduced by Khanfar [9,14,16], provides a structured abstraction for source code representation. It captures the predicate of each conditional statement along with the statements within its main body. Elementary statements are represented as-is, while internal conditional statements are denoted as placeholders, $ph()$.

A PCB is formally defined as: $pcb ::= \{[b, s^1, \ldots, s^n], type, pcb\_parent\}$

Where a PCB consists of: (1) a predicate p; (2) a list of statements ([es^1, $ph(2)$, …, es^n]), where each (es^i) is an elementary statement and each (ph(i)) is a placeholder for an internal

[3] A path π is non-trivial if it contains at least two nodes, as defined in [15].

conditional or loop statement; (3) a type (type); and (4) a reference to its parent PCB, (pcb_parent). Additional information may be incorporated as needed.

The type *type* classifies PCBs as either *linear* (L), corresponding to conditional statements such as `if`, or *cyclic* (C), representing iterative constructs such as `while`.

The PCB representation preserves the syntactic structure of the program by maintaining a reference to its parent PCB, *pcb_parent*. The notation $\mathcal{P}(P_k)$ denotes the *pcb_parent* of P_k.

5 Theoretical Fundamentals

In structured source codes, identifying control dependencies within loops poses significant challenges primarily due to the presence of various jump statements such as `break`, `continue`, and `return`. Outside the loops, the complexity is reduced, as control dependencies arise only from the `if` conditions that control `return` or exit statements.

In this section, we establish some rules written as definitions, lemmas, theorems, and corollaries to properly deduce the right control dependencies from the syntax of the program statements.

5.1 Loop Follow Block

The syntax-directed approach for computing control dependencies identifies the labels of statements influenced by predicates based on their relative locations. The first relation location is the block of statements that are immediately next to the compound statements. Such blocks of labels are referred to as $P_\ell.follow$, where ℓ is the header label of the compound statement.

code 5: `void main(){if(b1`^{1}`)return`^{2}`;stm1`^{3}`;stm2`^{4}`;if(b2`^{5}`)return`^{6}`;stm3`^{7}`;stm4`^{8}`;}`

For example, in Code 5, we have three PCBs, which are: $P_(0)$={0,ph(1),3,4,ph(5),6,7}, P_1={1,2}, and P_4={4,5}. Additionally, it defines the next immediate blocks to P_1 and P_5 as: $P_1.follow$={3,4,5} and $P_5.follow$={6,7}.

Definition 7. *A loop follow block is the unique basic block (or node) in a control flow graph (CFG) that is executed immediately after the completion (termination) of a loop. It is the first basic block to be executed once the loop condition becomes false and the loop body is no longer executed.*

Inside the loops, the `if` conditional statements might control the execution of three different blocks. First, there is its follow block (Code 1, Label $4\xrightarrow{cd}$Label 2). Second, it refers to the header of the closest outer loop that exists in (Code 6, Label $6\xrightarrow{cd}$Label 8). Third, it pertains to the follow block of its closest outer loop (Code 1, Label $10\xrightarrow{cd}$Label 8).

code 6: `while(b`^{1}`){ if(b1`^{2}`)continue`^{3}`;if(b2`^{4}`)continue`^{5}`;`
`while(b3`$^{\eta=6}$`){cnt++`^{7}`;if(b4`^{8}`)return`^{9}`;}if(b5`^{10}`)continue`^{11}`; }`

5.2 Chained Controlling Predicates

In this context, we introduce the notion of *Chained Continue Predicates*—a sequence of predicates within a loop, where each predicate controls a `continue` statement as well as the execution of the next predicate in the sequence. In contrast to the preceding predicates, the final predicate in the sequence controls two `continue` statements. Formally, it can be expressed as follows:

Definition 8 (Chained Continue Predicates). *Let $P = [P_1, P_2, \ldots, P_n]$ be a sequence of predicates of* `if` *conditional statements inside the body of a loop, where each P_i ($1 \leq i < n$) controls a* `continue` *statement and P_{i+1}, while P_n controls two* `continue` *statements.*

As well, we define *Chained Break Predicates* and *Chained Controlling Return Predicates* as follows:

Definition 9 (Chained Break Predicates). *Let $P = [P_1, P_2, \ldots, P_n]$ be a sequence of predicates of* `if` *conditional statements inside the body of a loop, where each P_i $(1 \leq i < n)$ controls a* `break` *statement and P_{i+1}, while P_n controls two* `break` *statements.*

Definition 10 (Chained Return Predicates). *Let $P = [P_1, P_2, \ldots, P_n]$ be a sequence of predicates, where each P_i $(1 \leq i < n)$ controls a* `return` *statement and P_{i+1}, while P_n controls two* `return` *statements.*

Remark 1. If a predicate Q controls a *Chained Return Predicate* sequence $P = [P_1, P_2, \ldots, P_n]$, then Q is considered to control the entire sequence by controlling the first predicate P_1.

Remark 2. When referring to the label of a *chained predicate*, it denotes the label of the first predicate in the chain.

For example, in Code 7, there is a Chained `continue` Predicate consisting of the sequence [`b3`, `b4`, `b5`]. We refer to the label of the chain as 4, which corresponds to the label of `b3`. Since `b2` controls the execution of `b3`, we say that `b2` also controls the execution of the Chained `continue` Predicate [`b3`, `b4`, `b5`].

code 7: `while(b1`^{1}`)`$^{\circ}${`f1()`^{2}`;if(b2`^{3}`)`$^{\bullet}${`if(b3`^{4}`)continue`^{5}`;x=5`^{6}`;if(b4`^{7}`)continue`^{8}`;else`^{9}`f2()`^{10}`;`
`if(b5`^{11}`)continue`^{12}`;continue`^{13}`;`$^{\bullet}$`}f3()`^{14}`;`$^{\circ}$`}`

5.3 Lemmas

This section has general lemmas that would be used in proving some upcoming theorems.

Lemma 1 *Suppose ω is the header of a compound statement. There is a path from ω to each statement inside its body.*

Proof. Def. 4 states that there is a path from Entry to each statement in the code. Because all paths in structured code from Entry to any statement in a loop pass through its header, there must be a path from ω to each statement in its body. □

Lemma 2 *Suppose ω is the header of a loop, there is path exists from P_ω.follow to End that bypasses all statements in P_ω.*

Proof. Assume, for contradiction, that all paths from P_ω.*follow* to End must pass through ω. Because there is a structured flow from ω to ω.*follow*, this would imply an infinite loop, contradicting Def. 4, which requires the existence of a path from each label to End. □

5.4 General

Theorem 1. *Suppose ω is the predicate of a* `while` *loop that contains an internal* `if` *condition with the predicate η. If η controls the execution of two identical jump statements—whether two* `continue`, `break`, *or* `return` *statements—then η does not control either ω or P_ω.follow.*

Proof. If η controls two `continue` statements, all successor paths from η converge at ω, violating Def. 6-B. Thus, η controls neither ω nor $P_\omega.follow$. If η controls two `return` statements, its successor paths bypass both ω and $P_\omega.follow$, failing Def. 6-A. Consequently, η controls neither. If η controls two `break` statements, no path from η includes ω, and all paths converge at $P_\omega.follow$. Hence, η does not control $P_\omega.follow$ either. □

code 8: `while(b`^{1}`){if(b1`^{2}`)return`^{3}`;if(b2`^{4}`){if(b`^{5}`)continue`^{6}`;continue`^{7}`;}f1()`^{8}`;}f2()`^{9}`;`

In Code 8, suppose $\omega = 1$, $\eta = 5$, and $P_\omega.follow$ corresponds to Label 9. Then, η does not control Label 1 or Label 9.

Theorem 2. *Suppose η is the predicate of an* `if` *condition that controls exactly one jump statement. Then, $P_\eta.follow$ is control dependent on η.*

Proof. Since P_η encloses a jump statement, $P_\eta.follow$ necessarily post-dominates one of its successors, satisfying Def. 6-A. To satisfy Def. 6-B, a path must exist from one of η ' successors to End without passing through $P_\eta.follow$.

If the jump is a `return` statement, such a direct path exists, confirming that η controls $P_\eta.follow$. Otherwise, if the jump is a `continue` or `break` statement labeled ℓ, then P_η lies within a loop P_ω with ω as its header. The paths [η, ℓ, ω, $P_\omega.follow$] (for `continue`) and [η , ℓ, $P_\omega.follow$] (for `break`) exist. By Lemma 2, a path from $P_\omega.follow$ to End excludes ω and, accordingly, $P_\eta.follow$, satisfying both properties in Def. 6. Thus, $P_\eta.follow$ is control dependent on η. □

In Code 1, `b1` controls Labels 3 and 4, while `b2` controls Labels 7 and 8.

Corollary 1 *Suppose η is the predicate of an* `if` *condition that does not control any jump statement. Then, η does not control $P_\eta.follow$.*

Proof. At $P_\eta.follow$, the two paths from the successors of η converge. Therefore, Def. 6-B is not satisfied. □

Theorem 3. *Suppose η is the predicate of an* `if..else` *conditional statement, and each of its branches contains a jump statement. Then, η does not control $P_\eta.follow$.*

Proof. In structured programs, there are three jump statements: `break`, `continue`, and `return`. Each of these transfers control outside the loop associated with η. Since the control exits the loop, it implies that there exists a path to End that does not include $P_\eta.follow$. Therefore, under this assumption, Def. 6-A is not satisfied. □

Theorem 4. *Suppose ω is the header of a loop containing a compound statement P_η, either a loop or a conditional statement. If P_η contains a jump flow avoiding η and $P_\eta.follow$, then P_η prevents all prior predicates in P_ω from controlling $P_\eta.follow$.*

Proof. The jump from P_η leads to ω, $P_\omega.follow$, or End. According to Lemma 2, a path from $P_\omega.follow$ to End excludes all labels in P_ω. Additionally, there is direct program flow from ω to $P_\omega.follow$. In all three cases, a path from η to End exists without passing through $P_\eta.follow$. Consequently, no post-dominance exists between the previous predicates of η over $P_\eta.follow$, meaning Def. 6-B is not satisfied. □

In Code 1, `b1` controls Labels 4 and 5 but cannot control Label 7.

Theorem 5. *Assume $\mathcal{P}(P_k) = P_\eta$, where P_η contains a jump statement labeled ℓ, P_k contains a jump labeled ℓ', and $\ell > k$, then the predicate k does not control $P_\eta.follow$.*

Proof. The Def. 6-B between a successor of k and P_η.*follow* cannot be satisfied due to the presence of ℓ. □

code 9: `while(b`^{1}`){if(b2`^{2}`){if(b3`^{3}`)break`^{4}`;f1()`^{5}`;x=5`^{6}`;continue`^{7}`;}cnt++`^{8}`;};`

In Code 9, the predicate `b3` controls Labels 5, 6, and 7; however, it cannot control Label 8 because of `continue` at Label 7. If the statement at Label 7 is not a jump, then 8→3.break

5.5 Controlling The Header of the Loop

Theorem 6. *Suppose ω is the predicate of a* `while` *loop containing an internal* `if` *condition with predicate η. If η controls a* `continue` *statement, or controls a chained* `continue` *predicate, and there exists a path from η within the* `while` *loop to a* `break` *or* `return` *statement that bypasses η, then $\omega \rightarrow \eta$.*

Proof. Consider first the case where η controls a `continue` statement labeled ℓ, and there exists a path from P_η to a `break` or `return` statement labeled ℓ'. In this case, two paths exist: [η, ℓ, ω] and [η, ℓ', P_ω.*follow*, End] (Lemma 2 proves the existence of a path from P_ω.*follow* to End).

Since all paths from the first successor of η (i.e., ℓ) lead to ω, and there exists a path from its second successor to End that excludes ω, it follows that $\omega \rightarrow \eta$.

Now consider the case where η controls a chained `continue` predicate labeled ℓ. All jump statements in the chain converge at ω. Thus, semantically, there is no difference between controlling a single `continue` statement and controlling a chained `continue` predicate. In both cases, $\omega \rightarrow \eta$. □

In Code 10, Label 1→Label 2. and Label 1→Label 4. In Code 11, 1→5

code 10: `while(b`^{1}`){if(b1`^{2}`)continue`^{3}`;if(b2`^{4}`)continue`^{5}`;f1()`^{6}`;if(b3`^{7}`)break`^{8}`}f2()`^{9}`;`

code 11: `while(b`^{1}`){if(b2`^{2}`)return`^{3}`;if(b3`^{4}`){if(b4`^{5}`)break`^{6}`;continue`^{7}`;}f1()`^{8}`;}f2()`^{9}`;`

Corollary 2 *Suppose ω is the predicate of a* `while` *loop with an internal* `if` *condition whose predicate η controls a* `break`, `return` *statement, a chained break predicate, or a chained return predicate, each labeled ℓ. Then, η controls ω if it also controls a* `continue` *statement or a non-jump last statement in the loop.*

Proof. There are two paths from η : [η, ℓ, P_ω.*follow*, End], which continues to End without passing through ω (Lemma 2), and a second path leading to ω , either via a `continue` statement or through the loop's natural control flow, jumping from its last statement to ω. Since both conditions in Def. 6 are met, $\omega \rightarrow \eta$. □

In Code 12, 1→3 and 1→5.

code 12: `while(b`^{1}`){f()`^{2}`;if(b1`^{3}`){cnt++`^{4}`;if(b2`^{5}`)break`^{6}`;}f1()`^{7}`;}f2()`^{8}`;`

code 13: `while(b1`^{1}`){if(b2`^{2}`){ if(b3`^{3}`)break`^{4}`;if(b4`^{5}`)break`^{6}`;f4()`^{7}`;break`^{8}`;};};`

Corollary 3 *Suppose ω is the predicate of a* `while` *loop with an internal* `if` *condition whose predicate η controls a* `break` *or* `return` *statement labeled ℓ. Then, η controls ω if it also controls a chained* `continue` *predicate.*

Proof. Since all the paths in the chained `continue` predicate converge at ω, we can apply the proof of Corollary 2. □

5.6 IF-Break Controls the Follow of a Loop

Theorem 7. *Suppose ω is the label of a loop predicate containing an* `if` *condition with predicate η . The relation P_ω.follow→η holds if η controls a* `break` *statement inside P_ω , or a chained break predicate, and has an internal path within P_ω bypassing η and reaching a* `return` *statement.*

Proof. Suppose the label of the `break` statement or the chained break predicate is ℓ, and the label of the `return` statement is ℓ'. Further, assume that η has two successors, k and k'. Since η controls the execution of ℓ, it follows that ℓ must post-dominate one of its successors; assume this is k. This implies that P_ω.*follow* post-dominates `k`, thereby satisfying Def. 6-A.

Moreover, the predicate η can reach ℓ' only through its other successor, k', without exiting the loop. Therefore, there exists a path [η , k', ..., End] that does not include P_ω.*follow*. As a result, Def. 6-B is also satisfied, and we conclude that P_ω.*follow*→η . □

Corollary 4 *Suppose there is a loop with predicate label ω containing a* `return` *statement labeled ℓ and an internal* `if` *predicate η, which controls a* `break` *statement. If η has a path to a* `continue` *statement labeled ℓ' or to the last statement in the loop, then P_ω.follow→η , even if $\ell < \eta < \ell'$.*

Proof. The result follows directly from Theorem 7 by applying the same reasoning to the case where both a `break` and a `continue` statement are present and interact with the `return` behavior inside the loop. Therefore, no additional steps are needed. □

In Code 14, 10→2 and 10→4. In Codes 11, the predicate `b4` controls `f2()`.

code 14: `while(b`^{1}`)`$\overset{\circ}{\{}$`if(b1`^{2}`)break`^{3}`;if(b2`^{4}`)break`^{5}`;f1()`^{6}`;if(b3`^{7}`)`$\overset{\bullet}{\{}$`if(b4`^{8}`)return`^{9}`;`$\overset{\bullet}{\}}\overset{\circ}{\}}$`f2()`^{10}`;`

Corollary 5 *Suppose ω is the predicate of a* `while` *loop that contains an internal* `if` *condition whose predicate η controls the execution of either a* `return` *statement or a chained return predicate. Additionally, it also controls a* `break` *statement or a chained break predicate. Then, P_ω.follow→η.*

Proof. By Theorem 7, the claim holds under the same conditions by applying its reasoning to the current case. □

5.7 Loop Predicates Controlling their Follow Statements

Theorem 8. *If ω is the predicate of a* `while` *loop that includes a* `return` *statement, then P_ω.follow→ω holds.*

Proof. The label ω has two successors, one being P_ω.*follow*, satisfying the first condition of Def. 6, while the second path begins at ω+1. By Lemma 1, a path exists from ω to each statement inside its loop. An internal path from ω to a `return` statement within the loop implies a path from its second successor to End that excludes P_ω.*follow*, satisfying the second condition of Def. 6. Thus, P_ω.follow→ω. □

In Codes 11, and 15, Label 1 controls the execution of `f2()`. In Code 16, Label 7→Label 2.

code 15: `while(b`^{1}`){ if(b1`^{2}`)break`^{3}`;if(b2`^{4}`){ if(b3`^{5}`)return`^{6}`}} f2()`^{7}`;`

code 16: `while(b`^{1}`){while(b0`^{2}`){if(b1)return`^{3}`;counter++`^{4}`;}x=5`^{5}`;break`^{6}`;}f1()`^{7}`;`

6 Data Sets and Flags

In this approach, each PCB maintains seven datasets that the analysis propagates forward from the first to the last statement in the PCB. The first dataset, Ω_{int}, stores predicates definitively controlling the immediate next visited statement within the PCB. Two datasets, $\mathcal{E}_h$ and $\mathcal{C}_h$, contain predicates potentially controlling the loop header, while three others, $\mathcal{R}_f$, $\mathcal{C}_f$, and $\mathcal{B}_f$, track predicates potentially controlling the loop follow block ($P_\omega.follow$). The final dataset, Ω_{ext}, holds predicates that certainly control $P_\omega.follow$.

Datasets are divided into two categories: *Potential Controlling Predicates* ($\mathcal{C}_h$, $\mathcal{E}_h$, $\mathcal{C}_f$, $\mathcal{B}_f$, and $\mathcal{R}_f$), comprising predicates potentially controlling either the loop header or loop follow block by satisfying one property from Def. 6; and *Certain Controlling Predicates* (Ω_{int} and Ω_{ext}), consisting of predicates definitively controlling specific regions.

The Potential Controlling Predicates datasets are summarized as follows: (I) $\mathcal{E}_h(P_k)$ contains predicates controlling `break` or `return` statements, thus satisfying Def. 6-B in relation to the loop header; (II) $\mathcal{C}_h(P_k)$ includes predicates controlling `continue` statements, potentially controlling the loop header (satisfying Def. 6-A); (III) $\mathcal{B}_f(P_k)$ contains predicates controlling `break` , potentially controlling $P_\omega.follow$, satisfying Def. 6-A, while their satisfaction of Def. 6-B remains under investigation; (IV) $\mathcal{C}_f(P_k)$ comprises predicates either directly controlling or having paths to `continue` statements, potentially controlling $P_\omega.follow$, with both properties from Def. 6 still under verification; and (V) $\mathcal{R}_f(P_k)$ tracks predicates having paths to `return` , satisfying Def. 6-B regarding $P_k.follow$.

Each PCB also maintains three flags: $\nabla\mathcal{B}(P_k)$, indicating the presence of a `break` statement; $\nabla\mathcal{R}(P_k)$, marking the presence of a `return` statement; and $\nabla\mathcal{C}(P_k)$, indicating the existence of a `continue` statement.

The following subsections detail the visit function invoked for statements of type `if`, `while`, `break`, `return`, or `continue`.

6.1 Events and Actions

This work relies on propagating Potential Controlling Predicates within PCBs to identify or reject suspected control dependencies. During propagation, various *Events* occur, such as visiting a `return` statement or encountering an `if` PCB containing a `break` statement. Some of these events prompt decisions, referred to as *Actions* in this context.

The upcoming sections organize this process by grouping Events that trigger specific Actions for particular datasets. These Events include: `VST_RET` (visiting a `return` statement), `VST_BRK` (visiting a `break` statement), and `VST_CNT` (visiting a `continue` statement). Additional events are `VST_PCB_CNT`, `VST_PCB_BRK`, and `VST_PCB_RET`, which refer to visiting placeholders of PCBs containing a `continue` , `break` , or `return` statement, respectively.

Events signaling termination include: `EoI`, marking the end of an `if` condition; `EoL`, marking the end of a loop; `EoL_NR`, indicating the end of a loop without a `return` statement; and `EoL_R`, signifying the end of a loop containing a `return` statement.

Event actions are categorized into three types: $kill(\mathcal{E}_h(P_k))$, $gen\{\mathcal{E}_h(P_k)\}$, and $copy\{\mathcal{E}_h(P_k)\}$. The first action clears the dataset $\mathcal{E}_h(P_k)$. The second generally establishes a control dependency where the loop header becomes control dependent on each predicate within $\mathcal{E}_h(P_k)$; Lastly, reaching the end of a PCB may result in copying dataset contents into the corresponding dataset of its parent PCB.

The rules are structured clearly: Events appear on the left, followed by the notation $\vdash$ and the associated Action. Typically, detailed descriptions of actions are enclosed within $< * >$. Finally, implementation details for each Event $\vdash$ Action pair are explicitly noted using the notation $\triangleright$, pointing to the exact lines in the subsequent algorithms where these actions occur.

The notation "Alg. 2-6" refers to Algorithm 2, Line 6. Specifically, it denotes line 6 within Algorithm 2.

6.2 $\mathcal{E}_h$ - Predicates Exiting Loop and May Control Header of the Loop

This dataset implements Corollary 2. The dataset $\mathcal{E}_h(P_k)$ collects all predicates that have a path to a `break` or `return` statement, thereby satisfying Def. 6-B. As the analysis progresses inside the PCB P_k, if $\mathcal{E}_h(P_k)$ encounters a `continue` statement before being cleared due to the presence of a branch that prevents the satisfaction of Def. 6-A, then each predicate in this set is considered to control the header of the loop.

In Code 17, after processing $ph(3)$, $\mathcal{E}_h(P_2)$ becomes $\{3\}$. Upon processing P_5, the analysis clears $\mathcal{E}_h(P_2)$ and sets $\mathcal{B}_f(P_2)$ to $\{5\}$. Traversing Label 7 and $ph(8)$ leaves $\mathcal{E}_h(P_2)$ unchanged, as the predicates in $\mathcal{E}_h(P_2)$ do not directly control the `continue` statement at Label 9, but they have paths to it. At Label 10, the analysis encounters the `continue` statement, confirming that Predicate 5 controls Label 1.

Initially, $\mathcal{E}_h(P_2)$ contains predicates with paths to End excluding Label 1, thus potentially satisfying Def. 6-B relative to Label 1. Predicate 3 (`b3`) is removed during the processing of $ph(5)$ because, despite having a valid path to End, it does not satisfy Def. 6-A as Statement 6 directs the second successor's path to End. Visiting P_8 does not affect $\mathcal{E}_h(P_2)$, since P_8 neither satisfies Def. 6-A nor invalidates Def. 6-B at this point. However, at Label 10, the analysis verifies that Predicate 5 (`b4`) post-dominates all paths originating from it, thus fulfilling Def. 6-A in addition to Def. 6-B.

code 17: `while(b1`^{1}`)`$\overset{\circ}{\{}$`if(b2`^{2}`)`$\overset{\bullet}{\{}$`if(b3`^{3}`)break`^{4}`;if(b4`^{5}`)break`^{6}`;f()`^{7}`;if(b5`^{8}`)continue`^{9}`;`
`continue`^{10}`;`$\overset{\bullet}{\}}$`f1()`^{11}`;`$\overset{\circ}{\}}$`f2()`^{12}`;`

In the coming enumeration, we show the results of the propogation of $\mathcal{E}_h$ in the form of Event-to-Action Relationship.

Event $\vdash$ Action Rules for $\mathcal{E}_h$:

1. `VST_RET` $\vee$ `VST_BRK` $\vee$ `VST_PCB_RET` $\vee$ `VST_PCB_IF_BRK` $\vdash$ $kill(\mathcal{E}_h(P_k)) < \mathcal{E}_h(P_k)=\emptyset >$, $\triangleright$ Alg. 4-3, Alg. 5-7, Alg. 6-2, Alg. 7-8, Alg. 8-8, Alg. 8-17.
2. `VST_CNT` $\vee$ `EoL` $\vdash$ $gen\{\mathcal{E}_h(P_k)\} < \forall p \in \mathcal{E}_h(P_k), loop(P_k).header \rightarrow p >$ $\triangleright$ Alg. 3-3, Alg. 5-2,
3. `EoI` $\vdash$ $copy\{\mathcal{E}_h(P_k)\} < \mathcal{E}_h(P_k) \Rightarrow \mathcal{E}_h(\mathcal{P}(P_k)) >$ $\triangleright$ Alg. 7-12, Alg. 8-58, Alg. 8-69.

Reaching the end of a loop, meaning that, with the consideration of ***Killing Events***, no branch diverts the path from the predicate to the loop's end. This implies a direct path from one of the successors of the predicate in $\mathcal{E}_h(P_k)$ to $loop(P_k).header$. At this point, we conclude that each predicate in $\mathcal{E}_h(P_k)$ controls the execution of $loop(P_k).header$.

In Code 17, if we replace `f()` with `while(b){if(b)return`^{13}`;}`, then this prevents `b4` from controlling Label 1.

6.3 $\mathcal{C}_h$ Predicates Controlling Continue May Influence Loop Header

This dataset is designed to enforce Theorem 6. The dataset $\mathcal{C}_h(P_k)$ accumulates predicates that control `continue` statements. As the analysis advances within P_k, if this dataset encounters a path leading to a `return` or `break` statement, each predicate in $\mathcal{C}_h(P_k)$ is considered to control $loop(P_k).header$.

In Code 18, P_1 = [ph(2), ph(4), 13], and P_4 = [ph(5), ph(7), 9, ph(10)]. After processing P_2[4], visiting ph(2) yields $\mathcal{C}_h(P_1) = \{2\}$. Since P_4 contains a `break` statement, visiting ph(4) follows that 1→2. Furthermore, when the analysis reaches ph(10) within P_4, the content of $\mathcal{C}_h(P_4)$ is {4,5}. Consequently, visiting ph(10) establishes the control dependencies 1→5 and 1→7.

code 18: `while(b1`^{1}`)`$\overset{\circ}{\{}$`if(b2`^{2}`)continue`^{3}`;if(b3`^{4}`)`$\overset{\bullet}{\{}$`if(b4`^{5}`)continue`^{6}`;if(b5`^{7}`)continue`^{8}`;f1()`^{9}`;`
`if(b6`^{10}`)if(b7`^{11}`)break`^{12}`;`$\overset{\bullet}{\}}$`f2()`^{13}`;`$\overset{\circ}{\}}$`f3()`^{14}`;`

Event ⊢ Action Rules for $\mathcal{C}_h$:

4. `VST_CNT` ⊢ *kill*($\mathcal{C}_h(P_k)$)< $\mathcal{C}_h(P_k)$=∅ > ▷ Alg. 3-6
5. `VST_RET` ∨ `VST_PCB_RET` ∨ `VST_BRK` ∨ `VST_PCB_BRK` ⊢ *gen*{$\mathcal{C}_h(P_k)$}< ∀p ∈$\mathcal{C}_h(P_k)$, *loop*(P_k).*header*→p > ▷ Alg. 4-5, Alg. 5-4, Alg. 6-7, Alg. 7-6, Alg. 8-7, Alg. 8-15.
6. `EoI` ⊢ *copy*{$\mathcal{C}_f(P_k)$}< $\mathcal{C}_h(P_k)$⇒$\mathcal{C}_h(\mathcal{P}(P_k))$ > ▷ Alg. 7-22, Alg. 8-63.

6.4 $\mathcal{B}_f$ - Predicates Controlling Break May Influence the Loop Follow

This dataset is exploited to enforce Theorem 7. The dataset $\mathcal{B}_f(P_k)$ accumulates predicates that control `break` statements and chained break predicates. As the analysis proceeds within P_k, if this dataset encounters a `return` statement, all predicates within $\mathcal{B}_f(P_k)$ are considered to control *loop*(P_k).*follow*.

In Code 19, visiting ph(2) results in predicate 2 being added to $\mathcal{B}_f$(1). Since P_4 contains a `return` statement, visiting ph(4) establishes the control dependency 14→2. Within P_4, visiting ph(5) results in predicate 5 being added to $\mathcal{B}_f$(4). Similarly, visiting ph(7) adds predicate 7 to $\mathcal{B}_f$(4). Subsequently, visiting ph(10) establishes the control dependencies 14→5 and 14→7.

code 19: `while(b1`^{1}`)`$\overset{\circ}{\{}$`if(b2`^{2}`)break`^{3}`;if(b3`^{4}`)`$\overset{\bullet}{\{}$`if(b4`^{5}`)break`^{6}`;if(b5`^{7}`)break`^{8}`;f()`^{9}`;`
`if(b6`^{10}`)return`^{11}`;`$\overset{\bullet}{\}}$`if(b7`^{12}`)return`^{13}`;`$\overset{\circ}{\}}$`f2()`^{14}`;`

Event ⊢ Action Rules for $\mathcal{B}_f$:

7. `VST_CNT` ⊢ *kill*($\mathcal{B}_f(P_k)$)< $\mathcal{B}_f(P_k)$=∅ > ▷ Alg. 4-2.
8. `VST_CNT` ⊢ *copy*{$\mathcal{B}_f(P_k)$}< $\mathcal{B}_f(P_k)$⇒$\mathcal{W}_L$(*loop*(P_k)) > ▷ Alg. 3-2
9. `VST_RET` ∨ `VST_PCB_RET`, `EoL_R` ⊢ *gen*{$\mathcal{B}_f(P_k)$}< ∀p ∈$\mathcal{B}_f(P_k)$, *loop*(P_k).*follow*→p > ▷ Alg. 5-9, Alg. 6-5, Alg. 7-16, Alg. 8-57.
10. `EoI` ⊢ *copy*{$\mathcal{B}_f(P_k)$}< $\mathcal{B}_f(P_k)$⇒$\mathcal{B}_f(\mathcal{P}(P_k))$ > ▷ Alg. 7-27, Alg. 8-70.

6.5 $\mathcal{W}_L$ - Waiting List in Loop PCB

Suppose that we have the following scenario:

code 20: `while(b1`^{1}`)`$\overset{\circ}{\{}$`if(b2`^{2}`)return`^{3}`;if(b3`^{4}`)`$\overset{\bullet}{\{}$`if(b4`^{5}`)break`^{6}`;if(b5`^{7}`)continue`^{8}`;f1()`^{9}`;`
$\overset{\bullet}{\}}$`f2()`^{10}`;`$\overset{\circ}{\}}$`f3()`^{11}`;`

In Code 20, it is evident that 11→5 due to the existence of two distinct paths: [5,6,11], with no alternative path diverging from Label 6; and [5,7,8,1,End], which satisfies Def. 6-B.

[4] There is no need to confuse P_2 with ph(2). Processing P_2 means visiting all internal elements of P_2, whereas processing ph(2) refers to studying the effect of the findings inside P_2 on its parent, P_1.

However, consider replacing the statement `f1()` at Label 9 with a `break` statement. In this scenario, $\mathcal{B}_f(P_4)$ is cleared at Label 9, thus invalidating the control dependency between Label 11 and Predicate 5.

The complexity of this scenario arises from the presence of the `continue` statement at Label 8. Due to this statement, forward propagation cannot accurately detect preceding `return` statements appearing earlier in the code, even though these statements remain reachable from the `continue` statement.

To address this issue, a specialized dataset, $\mathcal{W}_L(P_k)$, is introduced at the loop PCB level. When the analysis encounters a `continue` statement or a PCB containing a `continue` statement, all predicates currently in $\mathcal{B}_f(P_k)$ are added to $\mathcal{W}_L(P_k)$. After processing all statements within the loop, the analysis verifies whether the loop includes any `return` statements. If such a statement is found, all predicates stored in $\mathcal{W}_L(P_k)$ are then considered to control the execution of $loop(P_k).follow$.

Applying this approach to Code 20, while processing $loop(P_3)$—corresponding to P_1—the analysis checks for the presence of a `return` statement by evaluating $\nabla\mathcal{R}(P_1)$. If a `return` statement exists, all predicates stored in $\mathcal{W}_L(P_1)$ are determined to control the execution of Label 9 or $P_1.follow$.

6.6 $\mathcal{C}_f$ - Predicates Reaching Continue May Influence the Loop Follow

The dataset $\mathcal{C}_f$ is created to implement Corollary 4. $\mathcal{C}_f$ collects all predicates that have paths to `continue` statements to later determine whether they also control a `break` statement. If this occurs, then when the analysis reaches the end of the loop, it checks whether the loop contains a `return` statement. If it does, then each predicate that controls a `break` statement and has a path to a `continue` statement is considered to control the follow block of the loop.

In Code 21, By tracking the paths, we find that predicates `b3`, `b4`, and `b5` control the execution of `f2()`. Taking `b4` as an example, the paths [5,8,9,10,12], [5,8,9,10,11,13], and [5,8,9,10,11,12,13] show that all paths starting from Label 8 (a successor of Predicate 5) reach Label 13. This implies that Label 14 post-dominates Label 8, thereby satisfying Def. 6-A . Additionally, Def. 6-B is satisfied by the path [5,6,7,1,2,3,End].

code 21: `while(b1`^{1}`)`$^{\circ}${`if(b2`^{2}`)return`^{3}`;if(b3`^{4}`)`$^{\bullet}${`if(b4`^{5}`)`$^{\diamond}${`if(b5`^{6}`)`$^{\times}${`continue`^{7}`;}`$^{\times\diamond\bullet}$`}}`
`f1()`^{8}`;if(b6`^{9}`)break`^{10}`;if(b7`^{11}`)break`^{12}`;break`^{13}`;`$^{\circ}$`}f2()`^{14}`;`

The dataset $\mathcal{C}_f$ shares the same challenge as $\mathcal{B}_f$. The presence of a preceding `return` statement, which is reachable through a `continue`, can disrupt the dependency analysis. Upon examining $\mathcal{C}_f(P_k)$, it becomes evident that it satisfies neither Def. 6-A nor Def. 6-B . Specifically, if a predicate in $\mathcal{C}_f(P_k)$ encounters a `return` statement, it does not control $loop(P_k).follow$. For example, In Code 22, the predicate `b3` does not control `f()` ($loop(P_4).follow$) since Predicate 4 never satisfies Def. 6-A with Label 7. However, replacing `x=5` with a `break` statement establishes a control dependency between `b3` and `f()` due to the two paths [4,6,7] (satisfying Def. 6-A) and [4,5,1,2,3,End] (satisfying Def. 6-B).

Thus, the primary role of $\mathcal{C}_f$ is to determine whether the predicates controlling a `continue` statement also control a `break` statement. If this occurs, the contents of $\mathcal{C}_f$ are transferred to $\mathcal{W}_L$, and the analysis continues until the end of the loop. If a `return` statement is encountered anywhere within the loop, all predicates stored in $\mathcal{W}_L$ are considered to control the loop's follow block.

code 22: `while(b1`^{1}`){if(b2`^{2}`)return`^{3}`;if(b3`^{4}`)continue`^{5}`;x=5`^{6}`;}f()`^{7}`;`

Event $\vdash$ Action Rules for $\mathcal{C}_f$:

11. VST_RET $\vee$ VST_CNT $\vee$ VST_PCB_RET $\vee$ VST_PCB_IF_CNT $\vee$ EoL $\vdash$ *kill*($\mathcal{C}_f(P_k)$)$<$ $\mathcal{C}_f(P_k)$=$\emptyset$ $>$ $\triangleright$ Alg. 3-7, Alg. 5-6, Alg. 6-4, Alg. 7-11, Alg. 8-16.
12. VST_BRK $\vdash$ *copy*{$\mathcal{C}_f(P_k)$}$<$ $\mathcal{C}_f(P_k)\Rightarrow\mathcal{W}_L(loop(P_k))$ $>$ $\triangleright$ Alg. 4-4
13. EoI $\vdash$ *copy*{$\mathcal{C}_f(P_k)$}$<$ $\mathcal{C}_f(P_k)\Rightarrow\mathcal{C}_f(\mathcal{P}(P_k))$ $>$ $\triangleright$ Alg. 7-14, Alg. 8-64

6.7 $\mathcal{R}_f$ - Predicates Reaching Return Statement May Control the Loop Follow

This dataset is used to implement Corollary 5. $\mathcal{R}_f(P_k)$ collects each predicate that controls a `return` statement or a chained return predicate. Later, if any of these predicates are found to also control a `break` statement or a chained break predicate, this implies that they control the execution of $loop(P_k).follow$.

Since these predicates satisfy Def. 6-B, satisfying Def. 6-A requires them to directly control a `break` statement. In turn, any branch from each of these predicates to `break` must be free from any jump statement or a PCB containing a jump statement.

code 23: `while(b1`^{1}`)`$\overset{\circ}{\{}$`if(b2`^{2}`)return`^{3}`;if(b3`^{4}`)return`^{5}`;while(b4`^{6}`)`$\overset{\bullet}{\{}$`cnt++`^{7}`;if(b5`^{8}`)return`^{9}`;`$\overset{\bullet}{\}}$
`x=1`^{10}`;if(b6`^{11}`)break`^{12}`;if(b7`^{13}`)break`^{14}`;break`^{15}`;`$\overset{\circ}{\}}$`f2()`^{16}`;`

In Code 23, Predicates 2 (`b2`) and 4 (`b3`) do not control `f2()`, whereas Predicate 6 (`b4`) does. This difference arises from the presence of a `return` statement positioned between predicates `b2`, `b3`, and the subsequent `break` statement at Label 15. In this scenario, we introduced a sequence of proxy `if` statements with `break`: (`if(b)break;if(b)break;...`).

The establishment of new dependency relations between predicates in $\mathcal{R}_f$ and the loop follow block depends on whether the propagation of $\mathcal{R}_f$ encounters a `break` statement. However, encountering a statement like `if(b)break;` neither satisfies Def. 6-A nor disrupts an already satisfied Def. 6-B; it remains neutral. Similarly, this neutral effect applies to datasets intended to capture predicates that satisfy Def. 6-B and seek satisfaction of Def. 6-A. Hence, the compound statement `if(b)continue;` has a neutral effect on the forward propagation of $\mathcal{E}_h$, and `if(b)break;` similarly has a neutral effect on the forward propagation of $\mathcal{C}_f$.

Event $\vdash$ Action Rules for $\mathcal{R}_f$:

14. VST_RET $\vee$ VST_PCB_RET $\vee$ VST_CNT $\vee$ VST_PCB_CNT $\vee$ EoL $\vdash$ *kill*($\mathcal{R}_f(P_k)$)$<$ $\mathcal{R}_f(P_k)$=$\emptyset$ $>$ $\triangleright$ Alg. 3-5, Alg. 5-8, Alg. 6-3, Alg. 7-10, Alg. 8-11, Alg. 8-18
15. VST_BRK $\vdash$ *gen*{$\mathcal{R}_f(P_k)$}$<$ $\forall p \in\mathcal{R}_f$,$loop(P_k).follow\rightarrow p$ $>$ $\triangleright$ Alg. 4-7
16. EoI $\vdash$ *copy*{$\mathcal{R}_f(P_k)$}$<$ $\mathcal{R}_f(P_k)\Rightarrow\mathcal{R}_f(\mathcal{P}(P_k))$ $>$ $\triangleright$ Alg. 7-13, Alg. 8-59

7 Algorithms

This approach is syntax-directed, using depth-first search (DFS) on the PCB-graph, ensuring traversal of program structures. When encountering a placeholder for a child PCB (P_k), a new set of datasets and flags is created in P_k to maintain isolation. The analysis then recursively explores all elements within P_k, applying appropriate visit functions based on statement types. Once the child PCB is fully analyzed, control returns to the parent PCB, resuming execution from where it left off. The datasets and flags in the parent PCB are updated either upon encountering a placeholder for a child PCB or when processing a jump statement (`return`, `break`, `continue`).

Algorithm 1: Main Visit Algorithm

```
1  VisitStm(ℓ, η, ω)
   Data: return
2  k: this integer datatype points to the next label in the program.
3  k= ℓ;
4  if η > 0 then ∀p ∈Ω_int(P_η), ℓ→p ;
5  switch ℓ.type do
6      case if:
7          Ω_int(P_ℓ) = ℓ;
8          k = VisitPCB(ℓ, ω) + 1;
9          if nextLabel.type ≠ else then
10             VisitIf(ℓ, η, ω)
11         else
12             Ω_int(P_k) = ℓ;
13             k = VisitPCB(k, ω) + 1;
14             VisitIfElse(ℓ, k, η, ω)
15         break;
16     case while ∨ for:
17         Ω_int(P_ℓ) = ℓ;
18         k = VisitPCB(ℓ, ω) + 1;
19         VisitWhile(ℓ, ℓ);
20         break
21     case break:    VisitBreak(ℓ, η, ω); break ;
22     case return:   VisitReturn(ℓ, η, ω); break ;
23     case continue: VisitContinue(ℓ, η, ω); break ;
24 return k
```

Algorithm 2: Visit PCB

```
1  VisitPCB(η, ω)
   Data:
   handler: global or static variable that handles the C file;
   counter: number of statements, which assigned to global variable of each new statement ;
2  handler = HandleFile(fileName);
3
4  repeat
5      stmt = ReadStm(fileHander);
6      counter++;
7      stmt.label = counter;
8      ArrayStm+=stmt;
9      counter = VisitStm(counter,η,ω);
10 until statement = "}";
11 return counter
```

Although our program representation is based on the PCB-graph, we do not find it necessary to explicitly construct this data structure. Instead, by reading the statements one by one, we can determine the beginning and end of each compound statement.

7.1 Notations

Before proceeding, we introduce key notations—ℓ, ω, and η—used across all algorithms. ℓ represents the currently visited statement, η is the header of the compound statement enclosing ℓ, and ω is the closest outer loop enclosing P_η. For example, in Code 23, if the currently visited statement is Label 5, then $\ell = 5$, η=4, and ω=1. If $\ell = 7$, then η=6, and ω=6.

Additionally, *handler* is a global variable managing the C file, including intraprocedural programs. *counter* tracks the last statement number, and ReadStm reads the next statement from the buffer.

7.2 Visit Statements

This approach relies on recursively calling visit functions that implement depth-first search. Algorithm 2 presents the procedure that initiates this approach. The function ReadStm reads statements sequentially from the file[5] and uses Regular Expression matching to determine whether a statement is an opening curly brace, `while`, `for`, `if`, `else`, `break`, `return`, `continue`, or a closing curly brace. Other statements are processed superficially using a specialized small parser that identifies their types (e.g., integer declarations, procedure calls, pointer dereferencing) without deep analysis. However, the primary focus is on determining their tokens and identifying their controlling predicates. The internal details of this function are beyond the scope of this work, but it is important to clarify how the Abstract Syntax Tree is bypassed.

Algorithm 1 serves as the core of this approach, invoking the appropriate function for each statement based on its type. It begins at the procedure header and terminates at its end when the procedure is closed. For simplicity, we assume each compound statement starts with an opening curly brace "{" and ends with a closing curly brace "}".

In this solution, all statements are read and stored in an array of objects called *ArrayStm*. The variable *counter* is an integer that tracks the order of statements and predicates in the source code, with its current value serving as the label of the currently visited statement. For simplicity, ℓ is treated as an instance representing a statement or predicate. For example, instead of writing $ArrayStm[\ell].type$, we use $\ell.type$ or simply refer to it as the "*type of* ℓ".

In Line 3, we assign the value of ℓ to k. The variable k is used to determine the return value, which corresponds to the last statement processed by this procedure. It is also used in the `if` case to determine whether the `if` statement is followed by an `else` branch. The second line, Line 4, makes each statement control dependent on the last predicates stored in $\Omega_{int}(P_\eta)$, which control the follow regions in the parent PCB.

When it comes to jump statements (`break` , `return` , `continue`), each is processed through a dedicated procedure: VisitBreak, VisitReturns, and VisitContinue, respectively. These procedures analyze the effect of each corresponding jump statement on the datasets within its PCB (denoted by P_η), as well as on the datasets of its enclosing loop (denoted by P_ω).

Regarding compound statements (`if` , `else` , `while` , `for`), they are analyzed in two steps. The first step is handled by VisitPCB, which sequentially explores and visits the internal statements of the new PCB created to represent the compound block, and updates its datasets according to the internal jumps and branches. The second step involves one of three specific functions—VisitWhile, VisitIF, or VisitIfElse—each of which evaluates the impact of the corresponding compound statement on the datasets in its parent PCB (P_η) and its loop (P_ω).

The case of `if..else` requires special handling. In this case, two PCBs are created—P_ℓ and P_k—one for each branch. Thus, VisitPCB is called twice: once for each branch. For the second branch, which begins with the `else` keyword, ℓ is considered the first predicate controlling the

[5] In the actual implementation, this could be a buffer.

statements in its body, even though the PCB itself begins from the `else` keyword. Afterward, the procedure VisitIfElse is called.

7.3 Visit continue

Algorithm 3 illustrates the effect of visiting a `continue` statement with label ℓ, positioned at the end of the PCB P_η. Lines 9 adds η to $\mathcal{C}_h(P_\eta)$ and $\mathcal{C}_f(P_\eta)$ if no other child predicate within P_η controls a `break` or `return` statement. Line 10 enforces Theorem 5. Finally, since P_η contains a `continue` statement, the flag $\nabla\mathcal{C}(P_\eta)$ must be set (Line 11).

Algorithm 3: Visit a Continue Statement

1 VisitContinue(ℓ,η, ω)
2 $\mathcal{W}_L(P_\omega)$ += $\mathcal{B}_f(P_\eta)$; $\mathcal{B}_f(P_\eta)$=$\varnothing$;
3 $\forall p \in \mathcal{E}_h(P_\omega)$, $\omega \rightarrow p$;
4 $\mathcal{E}_h(P_\eta) = \varnothing$;
5 $\mathcal{R}_f(P_\eta) = \varnothing$;
6 $\mathcal{C}_h(P_\eta) = \varnothing$;
7 $\mathcal{C}_f(P_\eta)$=$\varnothing$;
8 **if** $\neg\nabla\mathcal{B}(P_\eta) \wedge \neg\nabla\mathcal{R}(P_\eta)$ **then**
9 | $\mathcal{C}_f(P_\eta)$ += η; $\mathcal{C}_h(P_\eta)$ += η
10 $\Omega_{int}(P_\eta)$=$\varnothing$;
11 $\nabla\mathcal{C}(P_\eta)$ = ***true***; $\nabla\mathcal{C}(P_\omega)$ = ***true***;

Algorithm 4: Visit a Break Statement

1 VisitBreak(ℓ,η, ω)
2 $\mathcal{B}_f(P_\eta) = \varnothing$;
3 $\mathcal{E}_h(P_\eta) = \varnothing$;
4 $\mathcal{W}_L(P_\omega)$ += $\mathcal{C}_f(P_\eta)$; $\mathcal{C}_f(P_\eta)$=$\varnothing$;
5 $\forall p \in \mathcal{C}_h(P_\omega)$, $\omega \rightarrow p$;
6 $\mathcal{C}_h(P_\omega) = \varnothing$;
7 $\Omega_f(P_\eta)$ + = $\mathcal{R}_f(P_\eta)$;
8 **if** $\neg\nabla\mathcal{C}(P_\eta) \wedge \neg\nabla\mathcal{R}(P_\eta) \wedge P_\eta$ *is* `if` **then**
9 | $\mathcal{E}_h(P_\eta)$ += η; $\mathcal{B}_f(P_\eta)$ += η
10 $\Omega_{int}(P_\eta)$=$\varnothing$;
11 $\nabla\mathcal{B}(P_\eta)$=***true***;

7.4 Visit break

Algorithm 4 outlines the steps performed when ℓ corresponds to a `break` statement. If ℓ is exclusively controlled by η, then Line 9 adds η to both $\mathcal{B}_f(P_\eta)$ and $\mathcal{E}_h(P_\eta)$. Furthermore, since P_η contains a `break` statement, Line 11 sets $\nabla\mathcal{B}(P_\eta)$. Finally, Line 10 clears $\Omega_{int}(P_\eta)$, in accordance with Theorem 5.

7.5 Visit return

Algorithm 6 outlines the steps performed when the currently visited statement, labeled ℓ , is a `return` statement. Since P_η has a `return` statement, Line 11 sets $\nabla\mathcal{R}(P_\eta)$. Line 9 adds η to $\mathcal{E}_h(P_\eta)$ and $\mathcal{R}_f(P_\eta)$ if and only Label η controls ℓ.

7.6 Visit while

The primary function of Algorithm 5, VisitWhile, is to update the datasets in the parent PCB, P_η, based on the internal contents of P_ℓ's datasets. A loop—whether a `while` or a `for`—establishes control dependencies outside itself only if it contains a `return` statement. Therefore, most of the statements in this procedure are executed only when the loop contains a `return` statement.

Line 3 checks for the presence of a `return` statement inside the loop, and if one is found, it executes additional steps that affect the datasets in the parent PCB (P_η).

Line 13 enforces Theorems 4, 8, and 2. Once all statements in P_ℓ are processed and visited, Line 14 adds all predicates in $\Omega_f(P_\ell)$ to $\mathcal{E}_h(P_\eta)$, as each already has a path to End through one of its successors. Since they control $P_\ell.follow$, if $P_\ell.follow$ terminates with a jump to the loop header, each predicate in $\Omega_f(P_\ell)$ also controls the loop header.

Since $P_\ell.follow$ is a successor of ℓ, their control dependence satisfies Def. 6-A. Moreover, as P_ℓ includes a `return`, it also satisfies Def. 6-B. Thus, Line 16 adds ℓ to $\Omega_f(P_\ell)$.

7.7 Visit if

According to Corollary 1, Line 3 checks whether the PCB has a jump statement. If not, it exits the procedure.

The propagation from child `if` PCBs to their parents is necessary because, unlike loops, which have a control flow returning from the last statement to the loop header, the paths and post-domination relationships of `if` successors remain unchanged due to the termination of the `if` condition. To illustrate this, consider the following example:

code 24: `while(b`^{1}`)`$\overset{\circ}{\{}$`if(b2`^{2}`)`$\overset{\bullet}{\{}$`if(b3`^{3}`)`$\overset{\triangledown}{\{}$`if(b4`^{4}`)continue`^{5}`;f1()`^{6}`;`$\overset{\triangledown}{\}}$`f2()`^{7}`;`$\overset{\bullet}{\}}$`break`^{8}`;`$\overset{\circ}{\}}$`f3()`^{9}`;`

Here, we can handle `f2()` similarly to `f1()` from `b4`'s perspective. Thus, the path [4,6,7,8] exists, and the end of the `if` conditional statement has no impact, as the transition occurs from an `if` statement to its parent, regardless of the parent's type.

Algorithm 5: Visit a `while` or `for` PCB

1 VisitWhileFor(ℓ, η, ω)
2 $\forall p \in \mathcal{E}_h(P_\ell)$, $\ell \rightarrow p$;
3 **if** $\nabla\mathcal{R}(P_\ell)$ **then**
4 $\quad \forall p \in \mathcal{C}_h(P_\eta)$, $\omega \rightarrow p$;
5 $\quad \mathcal{C}_h(P_\eta) = \varnothing$;
6 $\quad \mathcal{C}_f(P_\eta) = \varnothing$;
7 $\quad \mathcal{E}_h(P_\eta) = \varnothing$;
8 $\quad \mathcal{R}_f(P_\eta) = \varnothing$;
9 $\quad \Omega_f(P_\eta)$ += $\mathcal{B}_f(P_\eta)$;
10 $\quad \Omega_f(P_\ell)$ += ℓ;
11 $\quad \Omega_f(P_\ell)$ += $\mathcal{B}_f(P_\ell)$;
12 $\quad \Omega_f(P_\ell)$ += $\mathcal{W}_L(P_\ell)$;
13 $\quad \Omega_{int}(P_\eta) = \varnothing$; $\Omega_{int}(P_\eta)$ += $\Omega_f(P_\ell)$;
14 $\quad \mathcal{E}_h(P_\eta)$ += $\Omega_f(P_\ell)$;
15 $\quad \mathcal{R}_f(P_\eta)$ += $\Omega_f(P_\ell)$;
16 $\quad \Omega_{int}(P_\eta)$ += ℓ;
17 $\quad \nabla\mathcal{R}(P_\eta)$=***true***;
18 $\quad \nabla\mathcal{R}(P_\omega)$=***true***;

Algorithm 6: Visit a `return` Statement

1 VisitReturn(ℓ,η, ω)
2 $\mathcal{E}_h(P_\eta)=\varnothing$;
3 $\mathcal{R}_f(P_\eta)=\varnothing$;
4 $\mathcal{C}_f(P_\eta)=\varnothing$;
5 $\Omega_f(P_\eta)$ + = $\mathcal{B}_f(P_\eta)$;
6 $\mathcal{B}_f(P_\eta)=\varnothing$;
7 $\forall p \in \mathcal{C}_h(P_\eta)$, $\omega \rightarrow p$;
8 $\mathcal{C}_h(P_\eta) = \varnothing$;
9 **if** $\neg\nabla\mathcal{B}(P_\eta) \wedge \neg\nabla\mathcal{C}(P_\eta) \wedge \ell.type$ *is* `if` **then** $\mathcal{E}_h(P_\eta)$ += η; $\mathcal{R}_f(P_\eta)$ += η ;
10 $\Omega_{int}(P_\eta)=\varnothing$;
11 $\nabla\mathcal{R}(P_\eta)$=***true***;

Algorithm 7: Visit `if` Placeholder

1 VisitIF(ℓ, η, ω)
2 **if** $\neg\nabla\mathcal{B}(P_\ell) \wedge \neg\nabla\mathcal{R}(P_\ell) \wedge \neg\nabla\mathcal{C}(P_\ell)$ **then**
3 $\quad$ **return**
4 $\Omega_{int}(P_\eta)= \varnothing$;
5 **if** $\nabla\mathcal{B}(P_\ell) \vee \nabla\mathcal{R}(P_\ell)$ **then**
6 $\quad \forall p \in \mathcal{C}_h(P_\eta)$, $\omega \rightarrow p$;
7 $\quad \mathcal{C}_h(P_\eta) = \varnothing$;
8 $\quad \mathcal{E}_h(P_\eta) = \varnothing$;
9 **if** $\nabla\mathcal{C}(P_\ell) \vee \nabla\mathcal{R}(P_\ell)$ **then**
10 $\quad \mathcal{R}_f(P_\ell)=\varnothing$;
11 $\quad \mathcal{C}_f(P_\ell)=\varnothing$;
12 $\mathcal{E}_h(P_\eta)$ += $\mathcal{E}_h(P_\ell)$;
13 $\mathcal{R}_f(P_\eta)$ += $\mathcal{R}_f(P_\ell)$;
14 $\mathcal{C}_f(P_\eta)$ += $\mathcal{C}_f(P_\ell)$;
15 **if** $\nabla\mathcal{R}(P_\ell)$ **then**
16 $\quad \Omega_f(P_\eta)$ += $\mathcal{B}_f(P_\eta)$;
17 $\quad \mathcal{E}_h(P_\eta)$ += ℓ;
18 $\quad \mathcal{R}_f(P_\eta)$ += ℓ;
19 $\quad \nabla\mathcal{R}(P_\eta)$ = ***true***;
20 $\quad \nabla\mathcal{R}(P_\omega)$ = ***true***;
21 **if** $\nabla\mathcal{C}(P_\ell)$ **then**
22 $\quad \mathcal{C}_h(P_\eta)$ += $\mathcal{C}_h(P_\ell)$;
23 $\quad \mathcal{W}_L(P_\omega)$ += $\mathcal{B}_f(P_\eta)$;
24 $\quad \mathcal{C}_f(P_\eta)$ += ℓ;
25 $\quad \nabla\mathcal{C}(P_\eta)$=***true***;
26 **if** $\nabla\mathcal{B}(P_\ell)$ **then**
27 $\quad \mathcal{B}_f(P_\eta)$ += $\mathcal{B}_f(P_\ell)$;
28 $\quad \mathcal{E}_h(P_\eta)$ += ℓ;
29 $\quad \nabla\mathcal{B}(P_\eta)$ = ***true***;
30 $\Omega_{int}(P_\eta)$ += ℓ;
31 $\Omega_{int}(P_\eta)$ += $\Omega_{int}(P_\ell)$;
32 $\Omega_f(P_\eta)$ += $\Omega_f(P_\ell)$;

8 IF-Else Algorithm

Up to this point, the proposed work relies on propagating datasets to incrementally capture control dependencies by reading statements sequentially. This process involves capturing one of the dependency properties first, then propagating information to determine whether the second property is satisfied. This propagation works well with `if` and `while` compound statements, as they always have two successors: one inside the body of the compound statement, and the other in its follow region. By identifying where the body branch leads, and where the second successor flows and what it visits, control dependencies can be computed through sequential scanning.

However, this scenario does not apply to `if..else` statements, because they introduce two distinct execution paths. One path traverses all the statements in the first branch, while the other jumps from the `if` to the `else` and executes the statements in the second branch. The predicate is shared between both branches, and the first statement in each branch represents one of its successors. To check whether the control dependency properties are satisfied, a direct and explicit examination of both branches is required.

Alg. 8 is internally divided into three sections. The first spans from Line 6 to Line 19, and shows how visiting a `if` statement affects the datasets of its parent.

The second section, from Line 20 to Line 22, determines whether the predicate of the `if..else` statement controls the header or follow region of its loop. It also evaluates whether the predicate controls its own follow region. This depends on what exists in its two branches.

The final part, from Line 56 to the end, addresses what should be copied from both branches to the parent of the `if..else` statement. Although it is possible to summarize the algorithm, we chose to retain its full structure to make it easier to understand.

In this section, we focus primarily on the second part, as the first and third parts are already covered in Section 6.

The lines from 20 to 22 state that $\ell \rightarrow P_\omega.header$ if the predicate ℓ of a `if..else` statement exists in the $\mathcal{C}_h$ set of the PCB for one branch and reaches a `break` or `return` statement in the other branch. This satisfies Theorem 6.

code 25: `while(b`^{1}`)`$^{\circ}${`if(b2`^{2}`)`$^{\times}${`f1()`^{3}`;if(b1`^{4}`)continue`^{5}`;continue`^{6}`;`}$^{\times}$
`else`$^7_{\bullet}${`if(b2`^{8}`)`$^{\diamond}${`f2()`^{9}`;break`^{10}`;`}$^{\diamond}_{\bullet}$$^{\circ}$}}`f3()`^{11}`;`

In Code 25, 1→2 holds because Label 2 has two successors: 3 and 7, where Label 1 post-dominates Label 3 but does not post-dominate Label 7.

In Alg. 8. Line 20, it is worth noting that the dataset $\mathcal{C}_h$ contains the predicates that control either individual `continue` statements or chained continue predicate.

Lines 23 and 24 establish a control dependence relation $P_\omega.follow \rightarrow \ell$ if ℓ exists in the $\mathcal{B}_f$ dataset in one of the two branches and has a path to a `return` statement in the second branch. The predicate ℓ exists in $\mathcal{B}_f$ if it controls a `break` statement or a chained break predicate.

Lines from 25 to 30 add ℓ to $\mathcal{E}_h(P_\eta)$ if one of the branches of the `if..else` statement contains either a `break` or\and a `return` statement, and the other branch contains neither.

A natural question that might arise is: what happens if the second branch contains a `continue` statement? This scenario is handled in Lines 20 to 22.

The $\mathcal{C}_f$ set is designed to collect predicates that control both `continue` and `break` statements. If this occurs, the contents of $\mathcal{C}_f$ are copied into $\mathcal{W}_L$ to wait until the end of the loop event is reached. This procedure aligns with the Event–Action relationship described in Event#11 (page 17), where visiting a `return` or `continue` statement clears $\mathcal{C}_f$. Lines 32 and 36 implement these events by testing the two branches. If one of the branches P_ℓ contains a `continue` statement and the second branch P_k contains a `return` or `continue` statement, then it removes ℓ from $\mathcal{C}_f(P_\ell)$ (Line 32). The same test is applied to the second branch P_k(Line 36).

In Event#8 (Page 17), $\mathcal{B}_f$ is copied into $\mathcal{W}_L(P_\omega)$ if it encounters a `continue` statement. This applies to both branches of a `if..else` statement. Lines 34 and 38 implement this by adding ℓ to $\mathcal{W}_L(P_\omega)$ if one of the branches contains a `continue` statement while the other includes ℓ in its $\mathcal{B}_f$.

Lines 40 and 46 implement Event 15 (page 17), while Lines 44 and 50 implement Event 14 (page 17) at the level of the two branches within the currently visited `if..else` statement.

Line 51 implements Event#1(Page 14). Line 54 enforces Theorem 3.

Algorithm 8: Visit `if..else`

```
1  VisitIfElse(ℓ, k, η, ω)
2  if ¬∇B(P_ℓ)∧¬∇R(P_ℓ)∧¬∇C(P_ℓ)∧
3  ¬∇B(P_k)∧¬∇R(P_k)∧¬∇C(P_k) then
4  |  return
5  Ω_int(P_η)= ∅;
6  if ∇B(P_ℓ) ∨∇B(P_k) then
7  |  ∀p ∈ C_h(P_η), ω→p; C_h(P_η) = ∅;
8  |  E_h(P_η) = ∅;
9  |  Ω_int(P_ℓ) = ∅;
10 if ∇C(P_ℓ) ∨ ∇C(P_k) then
11 |  R_f(P_ℓ)=∅;
12 |  C_f(P_ℓ)=∅;
13 |  Ω_int(P_ℓ) = ∅;
14 if ∇R(P_ℓ) ∨∇R(P_k) then
15 |  ∀p ∈ C_h(P_η), ω→p; C_h(P_η) = ∅;
16 |  C_f(P_η) = ∅;
17 |  E_h(P_η) = ∅;
18 |  R_f(P_η) = ∅;
19 |  Ω_int(P_ℓ) = ∅;
20 if ( ℓ∈C_h(P_ℓ)∧(∇B(P_k)∨∇R(P_k)) )∨
21    ( ℓ∈C_h(P_k)∧(∇B(P_ℓ)∨∇R(P_ℓ)) ) then
22 |  ω→ℓ
23 if ℓ ∈B_f(P_ℓ)∧∇R(P_k) then Ω_f(P_ℓ)+=ℓ;
24 if ℓ ∈B_f(P_k)∧∇R(P_ℓ) then Ω_f(P_ω)+=ℓ;
25 if (∇B(P_ℓ)∨∇R(P_ℓ)) then
26 |  if ¬∇B(P_k)∧¬∇R(P_k) then
27 |  |  E_h(P_η)+=ℓ
28 if ∇B(P_k)∨∇R(P_k) then
29 |  if ¬∇B(P_ℓ)∧¬∇R(P_ℓ) then
30 |  |  E_h(P_η)+=ℓ
31 if ∇C(P_ℓ) then
32 |  if ∇C(P_k)∨∇R(P_k) then C_f(P_ℓ)-=ℓ ;
33 |  else C_f(P_ℓ)+=ℓ ;
34 |  if ℓ∈B_f(P_k) then W_L(P_ω)+=ℓ ;
35 if ∇C(P_k) then
36 |  if ∇C(P_ℓ)∨∇R(P_ℓ) then C_f(P_k)-=ℓ ;
37 |  else C_f(P_k)+=ℓ ;
38 |  if ℓ∈B_f(P_ℓ) then W_L(P_ω)+=ℓ ;
39 if ∇R(P_k) then
40 |  if ℓ∈B_f(P_ℓ) then Ω_f(P_ℓ)+=ℓ ;
41 |  else
42 |  |  if ¬∇C(P_ℓ)∧¬∇R(P_ℓ) then
43 |  |  |  R_f(P_k) += ℓ
44 |  |  else R_f(P_k) -= ℓ ;
45 if ∇R(P_ℓ) then
46 |  if ℓ∈B_f(P_k) then Ω_f(P_k) += ℓ ;
47 |  else
48 |  |  if ¬∇C(P_k)∧¬∇R(P_k) then
49 |  |  |  R_f(P_ℓ) += ℓ
50 |  |  else R_f(P_ℓ) -= ℓ ;
51 if (∇B(P_ℓ)∨∇R(P_ℓ))∧(∇B(P_k)∨∇R(P_k))
   then E_h(P_ℓ) -= ℓ; E_h(P_k) -= ℓ; ;
52 if (∇B(P_ℓ)∨∇R(P_ℓ)∨∇C(P_ℓ)) ∧
53 (∇B(P_k)∨∇R(P_k)∨∇C(P_k)) then
54 |  Ω_int(P_ℓ) -= ℓ; Ω_int(P_k) -= ℓ;
55 |  flagDual=true;
56 if ∇R(P_ℓ) ∨∇R(P_k) then
57 |  Ω_f(P_η) += B_f(P_η);
58 |  E_h(P_η)+=E_h(P_ℓ); E_h(P_η)+=E_h(P_k);
59 |  R_f(P_η)+=R_f(P_ℓ);R_f(P_η)+=R_f(P_k);
60 |  if ¬flagDual then E_h(P_η)+=ℓ;
   |  R_f(P_η)+=ℓ; ;
61 |  ∇R(P_ℓ)=true;
62 if ∇C(P_ℓ) ∨∇C(P_k) then
63 |  C_h(P_η)+=C_h(P_ℓ); C_h(P_η)+=C_h(P_k);
64 |  C_f(P_η)+=C_f(P_ℓ); C_f(P_η)+=C_f(P_k);
65 |  W_L(P_ω) += B_f(P_η)
66 |  if ¬flagDual then C_f(P_k) += ℓ;
67 |  ∇C(P_ℓ)=true;
68 if ∇B(P_ℓ) ∨∇B(P_k) then
69 |  E_h(P_η)+=E_h(P_ℓ);E_h(P_η)+=E_h(P_k);
70 |  B_f(P_η)+=B_f(P_ℓ); B_f(P_η)+=B_f(P_k);
71 |  if ¬flagDual then E_h(P_k) += ℓ;
72 |  ∇B(P_ℓ)=true;
73 Ω_int(P_η) += (Ω_int(P_ℓ)+Ω_int(P_k));
74 Ω_f(P_η) += (Ω_f(P_ℓ)+Ω_f(P_k));
```

9 Experimental Evaluations

File Size	Return Stmts	Break Stmts	Continue Stmts	If Stmts	While Stmts	Loop Depth
1K	42	84	78	249	86	7
5K	137	556	402	1,279	410	7
10K	272	1,080	884	2,572	846	7
20K	546	2,034	1,626	4,958	1,729	7
50K	1,462	5,185	4,190	12,624	4,236	7
75K	2,087	7,794	6,358	18,882	6,320	7
100K	2,539	8,372	7,751	22,417	8,668	7
150K	4,103	15,884	12,868	38,178	12,455	7

Table 1: Structural characteristics of generated test files.

In this section, we compare the performance of two methods for computing control dependencies: the state-of-the-practice method, which relies on control flow graphs and post-dominator trees, and our newly proposed approach, which utilizes control-flow equations. The experiments were conducted on a machine equipped with a 13th Gen Intel(R) Core(TM) i7-13,700HX processor (2.10 GHz) and 32 GB of RAM (31.6 GB usable). Both methods were implemented under identical conditions to ensure fairness. Specifically, both implementations utilized vectors as containers, maintained an array-of-structures memory layout, and operated under the same processor affinity and task priority (REALTIME_PRIORITY_CLASS). Additional optimizations were also applied to the standard method, such as pre-reserving vector capacities when constructing basic blocks. The test files used in these experiments were generated automatically. Each file represents an intraprocedural program, meaning that it contains only one procedure. Therefore, it is important to consider not only the file size but also structural aspects such as the number of `return`, `break`, `continue`, `if`, and `while` statements, as well as the depth of conditional statements and loops. These factors influence control dependencies and, consequently, the performance of the evaluated methods. Table 1 provides an overview of these structural characteristics across different file sizes.

Table 2: Execution time and Memory usages comparisons (Transposed)

Metric	1K	5K	10K	20K	50K	75K	100K	150K
New Method (μs)	155	457	801	1,510	3,571	5,311	6,666	10,399
Standard Method (μs)	309	1,651	3,989	11,871	56,632	118,944	176,709	448,733
Speedup Factor	1.99	3.61	4.98	7.86	15.87	22.40	26.52	43.16
New Method (KB)	3,480	4,132	4,912	6,504	11,256	15,108	18,824	26,716
Standard Method (KB)	4,076	5,548	6,144	8,784	14,980	20,384	25,460	36,996
Memory Reduction (%)	14.6%	25.5%	20.1%	25.9%	24.8%	25.9%	26.0%	27.8%

9.1 Execution Time & Memory Usage Comparisons

Table 2 presents the execution times (in microseconds) for both methods across various file sizes. The results indicate that the new method consistently outperforms the standard approach, demonstrating significant speed improvements, particularly for larger file sizes.

Table 2 shows the peak working set size (in KB) for both methods. Our proposed method not only runs faster but also consumes less memory than the standard approach, further demonstrating its efficiency.

9.2 Discussion

The standard method's execution times behave exponentially when the procedure size increases. This behavior arises from the necessity of constructing a large data structure to represent the nodes and flows of the control flow graph (CFG) and the post-dominator tree. This escalation is linked to enhanced memory caching and paging, contributing to latency and hardware limitations.

In contrast, the new method exhibits linear performance. This efficiency is attributed to four main reasons. First, its time complexity is strictly $O(N)$, ensuring that each statement is visited only once. Second, it does not construct any graph, avoiding the need to reserve and cache large memory blocks or spend time decoding and accessing memory locations. Although the philosophy of this work is based on the PCB graph, it does not explicitly build it. Instead, the natural alignment between the source code and the PCB graph enables the algorithms to replace graph construction with recursive procedure calls for each inner loop or conditional construct. Third, the new approach efficiently manages control flow by quickly creating and deleting datasets. This lightweight mechanism minimizes memory reservations and reduces latency. Lastly, the analysis method used to form and delete datasets for each compound statement enables dataset sharing, reducing the need to create unique datasets for each instance. This strategy significantly improves execution times and lowers memory usage.

In taking the standard method readings, this work deducts the time necessary to build the ASTs from the execution times. Additionally, it constructs Post-dominator trees over their corresponding Control Flow Graphs instead of treating them as two separate data structures. Consequently, these execution times reflect the cost of constructing a single graph compared to the three required in the classical method. As a result, we inadvertently biased the outcomes in favor of the classical method.

10 Related Work & History

The first attempt to develop a syntax-directed method for computing control dependencies was made by Ballance and Maccabe in 1992 in their work [11]. Harrold et al. later published two similar studies for the same purpose in 1993 [10] and 1996 [12]. In her second study, Harrold pointed out mistakes in the work of Ballance and Maccabe, while Han and Chen [13] later identified errors in Harrold's first study. In [12], Harrold's method relies on two definitions and two theorems to determine the predicates that control statements.

The first notable aspect of this work is its focus on simple cases, specifically those with no more than two jump statements within while loops. This simplicity is evident in the provided examples. Secondly, while the work presents a lengthy algorithm, it does not directly implement the theorems. Instead, it primarily displays the mathematical notations within the algorithm without explicitly applying them. Furthermore, the presentation lacks concrete examples that illustrate complex interactions between jump statements and predicates. Nevertheless, our focus here will remain on its theoretical background.

At the outset, Harrold defines the term *predicate path*, which is a sequence of predicates $\prod = \{P_n, ..., P_1\}$ where $\forall P_i, P_{i+1} \in \prod$, $P_i \rightarrow P_{i+1}$ holds.
Subseuqnetly, it defines the *control dependence negation* of $\prod$, denoted $\neg \prod$, is expressed as the following disjunction: $(P_n \wedge P_{n-1} \wedge ... \wedge P_{wp-1} \wedge \neg P_{wp}) \vee (P_n \wedge P_{n-1} \wedge ... \wedge \neg P_{wp-1}) \vee ... \vee (\neg P_n)$

where P_{wp}[6] is the first `while` predicate in the sequence from P_n to P_1.
Harrold then denotes the conjunction and disjunction of sets of predicate paths using the notations $\otimes$ and $\oplus$, respectively.
Subsequently, she introduces *Theorem 2*, which determines the control dependencies for a specific statement S_1 as follows:
If S is a `while` loop with predicate P and contains m jump statements, each controlled by predicate paths $\prod$, then the predicates controlling S.*follow* are determined based on the following cases:

- Case 1: if the jump statements do not include return statements, then there is no predicate controls S.*follow*.
- Case 2: If the jump statements consist of either `return` and continue statements, then S.*follow* is control dependent on $\otimes A$, where $A = \{\neg\prod_i \mid \prod_i$ is associated with some Ti that is a return statement$\}$.
- Case 3: If there are jump statements, including both `return` and `break` statements, then S.*follow* is control dependent on $\otimes A \vee \oplus B$, where: $A = \{\neg\prod_i \mid \prod_i$ is associated with some Ti that is a return statement$\}$, and $B = \{\prod_j \mid \prod_j$ is associated with some Tj that is a break statement$\}$.

This work explicitly states that it entirely disregards predicates enclosing `continue` statements when computing the control dependencies for S.*follow*. However, this is entirely incorrect. For instance, in Code 26, `b4` controls the execution of `stm2`, despite the fact that it controls a `continue` statement.

code 26:
```
while(b1){  if(b2)return;
            if(b3){if(b4)continue;if(b5)break;break;}stm1;  }
      stm2;
```

We sought a deeper understanding of how this theorem operates by analyzing its implementation. However, the implementation itself simply reiterates the same notations used in the theorem.
Nevertheless, when analyzing Code 8 and Code 27, we observe that, according to Harrold's theorem, they yield identical results. This, however, is incorrect. Specifically, in Code 8, we find that `f2()` $\xrightarrow{cd}$ `b1`, whereas this is not the case in Code 27.

code 27:
```
while(b1){if(b1)return;stm1} f1();
```

11 Conclusion

Our overarching research question is: *"How can control dependencies be computed in structured programs without constructing any type of graph?"* The key in answering this relies on tracking the two properties of control dependence relation separately through datasets associated with each compound statement. Based on this methodology, two key *requirements* must be met: the first is to visit each statement *only* once, and the second is to avoid storing any data beyond its immediate use, which is achieved by discarding the datasets of a compound statement once it is exited.

In comparing the proposed approach with the classical method—both designed for computing control dependencies in intra-procedural programs—we observe that the classical method treats all nodes equally, with analysis determined solely by control flows. It addresses this problem from a single dimension, focusing on *post-domination facts*, and attempts to resolve it through

[6] Since $n - 1 < n$, we believe the author intended to refer to P_{w+1} instead.

multiple stages, involving three layers of graphs, where each layer builds upon its predecessor. Consequently, while its algorithms are simple, they introduce significant computational overhead.

On the other hand, the proposed approach employs a *syntax-directed* method that successfully transforms combinations of jump and conditional statements from a post-domination perspective into a syntax-based representation. It achieves this by capturing various combinations of jumps and predicates through the creation of distinct datasets for each conditional statement or loop. The contents of these datasets are updated through a function that *kills* and *generates* dataset contents, which is invoked when visiting each predicate or jump statement. Although the primary philosophy behind these datasets is to create them at the beginning of each conditional statement or loop and remove them at the end, using a fixed number of datasets—without exceeding a certain limit—significantly enhances its performance.

Limitations or uncovered cases? This might be a philosophical question. However, the proposed approach in this work resolves the control dependence problem by manipulating a numerous number of different cases both individually and together. Individually, although the algorithm integrates different sets and flags, each subset is dedicated to handling specific types of cases. Together, they do not affect each other but share the same processing mechanism within the same procedures. Given the scope (structured code), the covered cases, and extensive experimental evaluations, we confirm that this approach does not currently have limitations. However, in the future, unresolved cases may emerge, though I believe this is unlikely. That said, I should emphasize that the included algorithms have undergone extensive testing.

This type of work requires setting up very fine configurations, which is done once but needs careful handling, as demonstrated in the presented algorithms. Any fault can produce a large number of errors. This presents a trade-off between using simple algorithms that operate in multiple stages, each accumulating results in memory, and employing a very fast solution for those who require such an approach.

The experimental evaluations demonstrate that the proposed syntax-directed approach yields correct results, exhibits scalability, and outperforms the classical approach. It achieves a performance improvement ranging from 2 to approximately 50 times that of the standard method.

Future work could explore optimizing the approach by reducing the number of datasets and minimizing the number of equations required when analyzing each jump or placeholder associated with a conditional statement or loop.

References

1. Karl J Ottenstein and Linda M Ottenstein. The program dependence graph in a software development environment. In *ACM Sigplan Notices*, pages 177–184. ACM, 1984.
2. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pages 319–349, 1987.
3. Ganesan Ramalingam. On loops, dominators, and dominance frontiers. *ACM transactions on Programming Languages and Systems*, 24(5):455–490, 2002.
4. Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the roman chariots problem. Technical report, Cornell University, 1997.
5. Paul Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.
6. Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. Technical report, Cornell University, 1994.
7. 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.
8. Thomas Ball and Susan Horwitz. Slicing programs with arbitrary control-flow. In *International Workshop on Automated and Algorithmic Debugging*, pages 206–222. Springer, 1993.
9. 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.
10. Mary Jean Harrold, Brian Malloy, and Gregg Rothermel. Efficient construction of program dependence graphs. *ACM SIGSOFT Software Engineering Notes*, 18(3):160–170, 1993.
11. Robert A. Ballance and Arthur B. Maccabe. Program dependence graphs for the rest of us. Technical report, University of New Mexico, 1992.
12. Mary Jean Harrold and Gregg Rothermel. Syntax-directed construction of program dependence graphs. *Technical Report OSU-CISRC-5/96-TR32*, 1996.
13. Zhe Han and Shihong Chen. A novel algorithm for construction control dependence subgraph. In *2009 International Conference on Multimedia Information Networking and Security*, volume 1, pages 158–162. IEEE, 2009.
14. Husni Khanfar and Björn Lisper. Enhanced PCB-based slicing. In *Fifth International Valentin Turchin Workshop on Metacomputation*, page 71, 2016.
15. 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.
16. Husni Khanfar, Björn Lisper, and Saad Mubeen. Demand-driven static backward slicing for unstructured programs. Technical Report MDH-MRTC-324/2019-1-SE, School of Innovation, Design and Engineering. Malardalen University, May 2019.
17. Husni Khanfar. Computing on-the-fly the relevant program flows to a control dependency. Technical Report MDH-MRTC-334/2021-1-SE, School of Innovation, Design and Engineering. Malardalen University, April 2021.
18. Husni Khanfar. *Overlapping Flows*. Number MDH-MRTC-349/2024-1-SE. Mälardalen Real-Time Research Centre, Mälardalen University, February 2024.