Semantical Equivalence of the Control Flow Graph and the Program Dependence Graph

Sohei Ito*
National Fisheries University, Japan

Abstract

The program dependence graph (PDG) represents data and control dependence between statements in a program. This paper presents an operational semantics of program dependence graphs. Since PDGs exclude artificial order of statements that resides in sequential programs, executions of PDGs are not unique. However, we identified a class of PDGs that have unique final states of executions, called deterministic PDGs. We prove that the operational semantics of control flow graphs is equivalent to that of deterministic PDGs. The class of deterministic PDGs properly include PDGs obtained from well-structured programs. Thus, our operational semantics of PDGs is more general than that of PDGs for well-structured programs, which are already established in literature.

1 Introduction

The program dependence graph (PDG) [9, 5, 6] is a kind of an intermediate representation of programs. It is a directed graph whose nodes are program statements and edges represent data dependence or control dependence between them.

The PDG is a useful representation for compiler code optimisation, because many optimisation techniques are based on analyses of data dependence and control dependence between statements in a program [4, 5]. Using PDGs, we can simplify program optimisations, because many optimisations can be carried out by simply scanning PDGs. If optimisation techniques are applied to control flow graphs (CFGs), we need to re-analyse dependence relationships in the program because optimisation changes them. However, if we apply optimisation techniques to PDGs, we do not need to re-analyse dependence relationships because optimisation itself changes them in the PDG. Furthermore, since the PDG excludes artificial order of statements, it is a suitable representation for vectorisation and parallelisation [10, 5, 2].

*Contact: ito@fish-u.ac.jp
Although the PDG is a suitable representation for program optimisation, the correctness of such optimisation techniques on PDGs is not well-studied due to the lack of research on semantics of the PDG.

There are a few researches on the semantics of PDGs [11, 3, 13]. Selke [11] introduced a rewriting semantics of a PDG and proved that it is equivalent to the program semantics. Cartwright and Felleisen [3] introduced a denotational semantics of a PDG. They proved that their semantics coincides with that of terminating programs, and also proved that even a PDG corresponding to a non-terminating program can return a result. Ramalingam and Reps [13] presented a formal semantics of program representation graphs (PRG) that are extension of PDGs. A PRG has so-called \( \phi \) nodes that determine which definition will be used when the same variable is defined in multiple assignment statements. Thanks to \( \phi \) nodes, PRGs can be naturally interpreted as data-flow programs. They define a semantics of a PRG as a function that maps an initial store to a sequence of values for each node, and is given as the least fixed point of a set of mutually recursive equations.

The above semantics of PDGs and PRGs cover only well-structured programs. Conditionals are restricted to the form (if \( e \) then \( . . . \) else \( . . . \)) and loops are restricted to the form (while \( e \) \( . . . \)). However, not all programs that appear during compiler code optimisation fall into this type of programs. Let us consider a program in Fig. 1 and assume that the expression \( x \times y \) is loop-invariant. Since this loop has a conditional that determines iteration, we cannot directly move the computation of \( x \times y \) outside of the loop. Therefore, we need to transform the loop and move the computation of \( x \times y \) before the loop as in Fig. 2 [10]. This loop has the form do \( . . . \) while and is left out of consideration in the existing semantics of PDGs. Therefore, existing semantics of PDGs are not sufficient to prove correctness of this optimisation technique.

In this paper, we present an operational semantics of PDGs that have more complex control flow structures than that of well-structured programs. In our operational semantics, statements that are independent of each other can be simultaneously executable. Hence, there are many executions for a single PDG. However, PDGs that satisfy certain structural constraints have the same exe-
conditional

T := x * y

z := T

conditional

Figure 2: After applying loop invariant code motion.

cution results. We call this type of PDGs deterministic PDGs (dPDGs).

Moreover, we prove that our operational semantics of PDGs coincide with that of CFGs for the class of dPDGs. It guarantees that if programs are semantically equivalent to CFGs, they are also semantically equivalent to PDGs. This enables us to use PDGs as a foundation to discuss correctness of several program transformations on PDGs.

This paper is organised as follows. In section 2, we introduce the CFG as a presentation of programs in this paper and its operational semantics. In addition to that, we define several types of dependence between statements in a CFG. In section 3, we define the PDG. Section 4 presents the operational semantics of PDGs. Furthermore, we define the class of deterministic PDGs and prove that PDGs constructed from usual programs are deterministic PDGs. In section 5, we prove that CFGs are semantically equivalent to the corresponding PDGs. The final section offers conclusion and future directions.

This paper is a revised version of the author’s previous works [8, 7] in that definitions are revised and proofs are optimised and rectified.

2 Control flow graph (CFG)

We consider the CFG as a representation of programs. A CFG is a pair \((N, E)\) of a set \(N\) of nodes and a set \(E \subseteq N \times N\) of edges. The nodes of a CFG are labelled by statements, and the edges of a CFG represent control flow of a program. We only consider three types of statements: an assignment statement \((x := e)\), a conditional statement \((\text{if } e)\) and a return statement \((\text{ret } x)\). Assignment statements have exactly one successor and conditional statements have exactly two successors. The edges from a conditional statement are labelled differently from each other, that is, either with true or false. A CFG has a unique start
node and a unique end node. The start node has no predecessor, while the end
node has no successor. Any node in a CFG is reachable from the start node.
The last node must be a return statement.

Fig. 3 is an example of a CFG.

**Definition 2.1.** Let \( \text{Var} \) be the set of variables and \( \text{Exp} \) be the set of expressions
appearing in a CFG. Let \( \text{val} \) be the set of values, which contains special values
\( \text{T} \) and \( \text{F} \). A store is a function of the signature \( \text{Var} \rightarrow \text{Val} \). We write \( \text{Store} \) for the set of stores. An
evaluation function of expressions is \( \text{E} : \text{Exp} \rightarrow (\text{Store} \rightarrow \text{Val}) \).

**Definition 2.2.** (The operational semantics of CFG) Let \( G = (N, E) \) be a
CFG. A run is a sequence of stores \( \sigma_0 \xrightarrow{n_0} \sigma_1 \xrightarrow{n_1} \ldots \) satisfying the following:

1. \( \sigma_0 \) is a given initial store.
2. \( n_0 \) is the initial node of \( G \).
3. \( n_{i+1} \) is a successor of \( n_i \) in \( G \) and determined as follows:

   (a) If \( n_i \) is an assignment statement, then \( n_{i+1} \) is the unique successor
       of \( n_i \).

   (b) If \( n_i \) is a conditional statement (\( \text{if} \ e \)), and \( (n_i, p) \in E \) and \( (n_i, q) \in E \), then

       \[
       n_{i+1} = \begin{cases} 
       p & \text{if } \text{E}(e)\sigma_i = \text{T} \\
       q & \text{if } \text{E}(e)\sigma_i = \text{F}
       \end{cases}
       \]

4. \( \sigma_{i+1} \) is determined as follows:

       \[
       \sigma_{i+1} = \begin{cases} 
       \sigma_i[x \mapsto \text{E}(e)\sigma_i] & \text{if } n_i = (x := e) \\
       \sigma_i & \text{otherwise}
       \end{cases}
       \]

where \( \sigma[x \mapsto v] \) is the same as \( \sigma \) except it maps \( x \) to \( v \).
The following theorem is a trivial consequence of the definition.

**Theorem 2.1.** A run of a CFG is deterministic, that is, for any CFG $G$ and $\sigma_0$ there exists the unique run of $G$.

There are several types of dependence between program statements. In the following sections, we define control dependence, data dependence and def-order dependence.

### 2.1 Control flow dependence

To define control dependence, we first define the notion of post-domination. We add entry node and exit node to a CFG. Entry node is the predecessor of the start node and exit node is the successor of the end node. Entry node has true edge to the start node and false edge to exit node. We call the resulting graph the augmented CFG of an original CFG. Figure 3 is the augmented CFG for the graph in Figure 4.

A path is a sequence of nodes $n_0n_1n_2\ldots$ where $\forall i \geq 0, (n_i, n_{i+1}) \in E$. The length of a path is the length of the sequence.

**Definition 2.3 (Post-domination).** Let $s$ and $t$ be nodes in a CFG $G$. We say $t$ post-dominates $s$ iff every path from $s$ contains $t$, and $t$ strictly post-dominates $s$ iff $t$ post-dominates $s$ and $s \neq t$.

**Definition 2.4 (Control dependence).** Let $s$ and $t$ be nodes in a CFG $G$. We say $t$ is control-dependent on $s$ iff there exists a path of length greater than 1 from $s$ to $t$ such that $t$ post-dominates all nodes on that path except $s$, and $t$ does not strictly post-dominate $s$.

The above definitions are slightly modified from usual post-domination and control dependence. The definition of post-domination and control dependence in this paper are called “strong forward domination” and “weak control dependence” [12]. The usual definition is “$s$ post-dominates $t$ if every path from $t$ to the exit node contains $s$”. Our definition omits the underlined phrase.

The difference between the usual definition and our definition appears when a CFG contains a loop. We illustrate this difference using an example CFG in Fig. 4. In this CFG, node 5 is not control-dependent on node 4 in the usual definition since every path from node 4 to the exit node contains node 5, i.e. node 5 post-dominates node 4. However, in our definition, node 5 is control-dependent on node 4 since every path from node 4 does not contain node 5. There exists a path iterating the loop infinitely. Thus, node 5 does not post-dominate node 4. Our definition reflects the intuition that whether node 5 is executed or not depends on the result of condition at node 4.

We discriminate two types of control dependence; true control dependence and false control dependence. Suppose $t$ is control-dependent on $s$. By definition, $s$ has two successors i.e. $s$ is a conditional statement. Let $(s,u)_{T} \in E$ and $(s,v)_{F} \in E$. If $u$ is post-dominated by $t$, $t$ is true control-dependent on $s$. If $v$ is post-dominated by $t$, $t$ is false control-dependent on $s$.

We write $CD(s,t)$ to represent that $t$ is control dependent on $s$. 


2.2 Data dependence

There are two kinds of data dependence – loop-independent data dependence and loop-carried data dependence. To define these types of dependence, we must define loops formally. We cannot use the definition of natural loops since we treat irreducible control flow graphs.

Definition 2.5 (Loops, back edges). Let $G$ be a directed graph. A loop is a strongly connected region in $G$. A back edge of a loop is an edge contained in that loop whose target node have an incoming edge from the outside of that loop.

In Fig. 5 the set of nodes \{1, 2, 3, 4, 5, 6\} comprises a loop. Back edges for this loop are (3, 4) and (6, 1). In Fig. 6 the set of nodes \{1, 2, 3, 4, 5\} and \{2, 3, 4\} comprise loops. The back edges for \{1, 2, 3, 4, 5\} and \{2, 3, 4\} are respectively (5, 1) and (4, 2).
Definition 2.6 (Data dependence). Let $G$ be a CFG. Let $s$ and $t$ be nodes in $G$. We say the definition at $s$ reaches $t$ if there exists a variable $w$ such that $s$ defines $w$ and $t$ uses $w$ and there exists a path whose length is more than 1 in $G$ from $s$ to $t$ such that no node except $s$ on that path defines $w$. We call such a path a reaching path from $s$ to $t$. We say $t$ is data-dependent on $s$ if the definition at $s$ reaches $t$. There is a loop-independent data dependence from $s$ to $t$ if both $s$ and $t$ are not contained in the same loop or ii) both $s$ and $t$ are contained in the same loop, there is a reaching path from $s$ to $t$ that does not contain any back edge of the loop. There is a loop-carried data dependence from $s$ to $t$ if both $s$ and $t$ are contained in the same loop, and there exists a reaching path from $s$ to $t$ that contains a back edge of the loop and there is no loop-independent data dependence from $s$ to $t$.

We write $LIDD(s, t)$ and $LCDD(s, t)$ to represent there is a loop-independent data dependence from $s$ to $t$ and there is a loop-carried data dependence from $s$ to $t$, respectively.

2.3 Def-order dependence

Finally we define def-order dependence.

Definition 2.7 (Def-order dependence). Let $G$ be a CFG. Let $s$ and $t$ be nodes in $G$. We say $t$ is def-order-dependent on $s$ if there exists a node $u$ which is loop-independently data dependent on both $s$ and $t$, and both $s$ and $t$ define the same variable, and i) there exists a path from $s$ to $t$ and both $s$ and $t$ are not contained in the same loop, or ii) both $s$ and $t$ are contained in the same loop and $t$ is reachable from $s$ without passing any back edge of the loop, or iii) if both $s$ and $t$ are contained in the same loop, and $s$ is reachable from $t$ and $t$ is also reachable from $s$, then such paths must contain back edges of the loop.

Note that in the case of iii), there is also a def-order dependence from $t$ to $s$. We write $DefOrd(s, t)$ to represent $t$ is def-order-dependent on $s$.

3 Program dependence graph (PDG)

The definition of the PDG in this paper depends on the one by Selke [11]. A PDG is a quintuple $(N, C, F, L, D)$ where $N$ is a set of nodes, and $C$, $F$, $L$ and $D$ sets of edges, that is, subsets of $N \times N$. The sets of edges $C$, $F$, $L$ and $D$ respectively represent four types of dependence: control flow dependence, loop-independent data dependence, loop-carried data dependence and def-order dependence. $C$-edges are divided into true edges and false edges.

The subgraph $(N, C)$ is called the control dependence graph (CDG) of a PDG $(N, C, F, L, D)$.

Nodes are labeled with statements as in CFGs. Statements are one of the following: an assignment statement ($x := e$), a conditional statement ($if e$) or a return statement ($return$).
If a node is labeled with an assignment statement, then there is no $C$-edges outgoing from that state. If a node is labeled with a conditional statement, then there is no outgoing $F$, $L$ and $D$-edges from that node. If $(p, q) \in F$ or $(p, q) \in L$, there is a variable $w$ that $p$ defines and $q$ uses. If $(p, q) \in D$ then there is a variable $w$ such that both $p$ and $q$ define it and there is a node $u$ such that $(p, u) \in F$ and $(q, u) \in F$. Nodes labeled with ret statements do not have any outgoing edges.

3.1 Construction of the PDG from a CFG

Let $G = (N, E)$ be a CFG. The PDG corresponding to $G$ is a graph $(N \cup \{entry\}, C, F, L, D)$, where, supposing $s$ and $t$ range over $N \cup \{entry\}$,

\[
C = \{(s, t) \mid CD(s, t)\}
\]
\[
F = \{(s, t) \mid LIDD(s, t)\}
\]
\[
L = \{(s, t) \mid LCDD(s, t)\}
\]
\[
D = \{(s, t) \mid DefOrd(s, t)\}
\]

Note that the node entry is the root node of the CDG $(N \cup \{entry\}, C)$.

Hereafter, we write $c(s, t)$ for an element of $C$, $f(s, t)$ for an element of $F$, $l(s, t)$ for an element of $L$ and $d(s, t)$ for an element of $D$. Furthermore, we discriminate true control dependence and false control dependence in $C$ by writing $ct(s, t)$ and $cf(s, t)$. We also discriminate elements in $F$ and $L$ by explicating a related variable like $fx(s, t)$ and $lx(s, t)$. Fig. 7 presents the PDG and its CDG corresponding to the CFG in Fig. 3.

4 An operational semantics of the PDG

In this section, we present an operational semantics of the PDG. The basic idea is to define states in runs of PDGs as follows.

\[
state = (avail, econf)
\]
**avail** is a function that takes a node and a variable and return a value. Recall that **store** is a set of functions taking a variable and return a value. The difference between **store** and **avail** is that **store** is global bindings of variables while **avail** is local bindings of variables for each node. We do not need global stores since an execution of a statement affects only nodes that are dependent on it.

**econf** is a function that takes an edge and return the state of the edge. Functions in **econf** indicate whether nodes are executable or not. For example, if node $n$ is an **if** statement and the expression is evaluated to $T$ then its outgoing $ct$-edges are “activated” and outgoing $cf$-edges are “inactivated.” Whether a node is executable or not is determined by the state of its incoming edges. **econf** is used to represent a configuration of edges in executions of PDGs.

To define the operational semantics of the PDG, we introduce the following notions on PDGs.

### 4.1 Preliminary

In order to define the operational semantics of the PDG, we define several notions related to it.

**Definition 4.1** (Looping edges). Let $G = (N, C, F, L, D)$ be a PDG and $G_C = (N, C)$ be the CDG of $G$. Let $R$ be a loop in $G_C$. Suppose $ct(p, q)$ (resp. $cf(p, q)$) is a back edge of $R$. The looping edges of $R$ are all $ct$-edges (resp. $cf$-edges) from $p$ whose destinations have predecessors other than $p$.

Note that the notions of loops and back edges are not restricted to the CFG. In the CDG in Figure 7 there is a loop which consists of only node 4. Its back edge is $ct(4, 4)$. Therefore, its looping edges are $ct(4, 2), ct(4, 3)$ and $ct(4, 4)$.

We explain the correspondence between loops in a CFG and those in the CDG. Before that, we define iteration statements of loops in the CFG.

**Definition 4.2** (Iteration statements). An iteration statement of a loop in a CFG is an **if** statement in that loop that has outgoing edges to nodes both inside and outside of that loop. It is trivial that for any loop in a CFG, there is at least one iteration statement.

Compare the CFG in Fig. 5 with its corresponding CDG in Fig. 8. We see that there is a loop $\{1, 2, 3, 4, 5, 6\}$ in Fig. 5 whose back edges are $\{3, 4\}$ and $\{6, 1\}$. Therefore, node 3 and 6 are iteration statements in this program. Meanwhile, the loop existing in Fig. 8 is $\{3, 6\}$. Thus, we can see that iteration statements in a CFG will be nodes in a loop in the corresponding CDG. Furthermore, the back edges of the loop $\{3, 6\}$ in Fig. 8 are $ct(3, 6)$ and $ct(6, 3)$. Therefore, the looping edges are $ct(3, 4), ct(3, 5), ct(3, 6), ct(6, 1), ct(6, 2)$ and $ct(6, 3)$. If a node in a loop in a CDG is an iteration statement, then the looping edges outgoing from that node are $C$-edges to the nodes that will be executed when the loop iterates.

Intuitively, looping edges represent control dependence to the statements that will be executed in the next step of the loop.
Definition 4.3. Let $G = (N, C, F, L, D)$ be a PDG. We define $\hat{C}$ as the set obtained by removing all looping edges from $C$.

Definition 4.4. Let $p$ and $q$ be nodes in a PDG $(N, C, F, L, D)$. We say $q$ is $C$-reachable from $p$ if $q$ is reachable from $q$ by $C$-edges. We say $q$ is $\hat{C}$-reachable from $p$ if $q$ is reachable from $q$ by $\hat{C}$-edges. We respectively say a $C$-successor, a $ct$-successor and a $cf$-successor for a successor by a $C$-edge, a $ct$-edge and a $cf$-edge.

Definition 4.5 (Subgraphs). Let $G = (N, C, F, L, D)$ be a PDG and $p \in N$. $G(p) = (N', C', F', L', D')$ is the subgraph of node $p$ defined as follows:

\[
\begin{align*}
N' &= \{ n \mid \exists q. c(p, q) \in C \land n \text{ is } \hat{C}\text{-reachable from } q \} \\
C' &= C \cap (N' \times N') \\
F' &= F \cap (N' \times N') \\
L' &= L \cap (N' \times N') \\
D' &= D \cap (N' \times N')
\end{align*}
\]
$G(p)$ is the subgraph that consists of nodes reachable from $p$ by $C$-edges that are not looping edges. Note that $c(p, q)$ itself may be a looping edge. Similarly, we define $G_T(p)$ (resp. $G_F(p)$) as the subgraph consisting of the nodes $\bar{C}$-reachable from $ct$-successors (resp. $cf$-successors) of $p$. Formally, $G_T(p) = (N', C', F', L', D')$ is defined as:

\[
\begin{align*}
N' &= \{ n | \exists q. ct(p, q) \in C \land n \text{ is } \bar{C}\text{-reachable from } q \} \\
C' &= C \cap (N' \times N') \\
F' &= F \cap (N' \times N') \\
L' &= L \cap (N' \times N') \\
D' &= D \cap (N' \times N')
\end{align*}
\]

The definition of $G_F(p)$ is obtained from replacing the condition $ct(p, q) \in C$ with $cf(p, q) \in C$. We further define $G^*(p) = (N', C', F', L', D')$ as follows:

\[
\begin{align*}
N' &= \{ n | \exists q. c(p, q) \in C \land n \text{ is } C\text{-reachable from } q \} \\
C' &= C \cap (N' \times N') \\
F' &= F \cap (N' \times N') \\
L' &= L \cap (N' \times N') \\
D' &= D \cap (N' \times N')
\end{align*}
\]

We can similarly define $G_T^*(p)$ and $G_F^*(p)$.

Intuitively, $G(p)$ consists of such statements that are at the same iteration level in the corresponding CFG. $G_T(p)$ consists of statements that will be executed when $p$ is evaluated to $T$, and $G_F(p)$ consists of statements that will be executed when $p$ is evaluated to $F$.

Fig. 8 shows subgraphs of the CDG in Fig. 5, where looping edges are $ct(3, 4)$, $ct(3, 5)$, $ct(3, 6)$, $ct(6, 1)$, $ct(6, 2)$ and $ct(6, 3)$.

Using above notions, we define the operational semantics of PDGs.

**Definition 4.6.** An avail and an econf are respectively members of the following sets of functions $\text{Avail}$ and $\text{Econf}$:

\[
\begin{align*}
\text{Avail} &= N \times \text{Var} \rightarrow \text{Val} \\
\text{Econf} &= C \oplus F \oplus L \oplus D \rightarrow \{ \text{chk}, \text{unchk}, \text{act} \}.
\end{align*}
\]

Here $\oplus$ is the direct sum. An evaluation function $\mathcal{E}$ is defined as

\[
\mathcal{E} : N \times \text{Exp} \rightarrow (\text{Avail} \rightarrow \text{Val}).
\]

A state is a member of $\text{State} = \text{Avail} \times \text{Econf}$.

Although we use the same symbol $\mathcal{E}$ as the evaluation function in definition 2.1, they are easily discriminated by the signatures.
Definition 4.7 (Executable nodes). We first introduce the following predicates. Suppose $s = (av, ec)$, $av \in \text{Avail}$, $ec \in \text{Econf}$ and $n$ is a node.

\[
\begin{align*}
\text{condC}(s, n) & \overset{\text{def}}{=} \exists (p, n) \in C. ec(c(p, n)) = \text{act} \\
\text{condF}(s, n) & \overset{\text{def}}{=} \forall f(p, n) \in F. ec(f(p, n)) = \text{chk} \\
\text{condL}(s, n) & \overset{\text{def}}{=} \forall l(n, p) \in L. n \neq p \Rightarrow ec(l(n, p)) = \text{chk} \\
\text{condD}(s, n) & \overset{\text{def}}{=} \forall d(p, n) \in D. ec(d(p, n)) = \text{chk}
\end{align*}
\]

We write $\text{condCFLD}(s, n)$ for $\text{condC}(s, n) \land \text{condF}(s, n) \land \text{condL}(s, n) \land \text{condD}(s, n)$.

We define function $\text{Next} : \text{State} \rightarrow 2^N$ as:

\[\text{Next}(s) = \{ n \mid \text{condCFLD}(s, n) \} .\]

$\text{Next}(s)$ represent the executable nodes at state $s$.

Definition 4.8 (Update function of avails). We define function $\text{udav} : N \times \text{Avail} \rightarrow \text{Avail}$ as follows:

\[\text{udav}(n, av) = \begin{cases} 
av((p, x) \mapsto E(n, e) av : (n, p) \in F \oplus L] & \text{if } n = (x := e) \\
av & \text{otherwise}
\end{cases}\]

where $\text{udav}((p, x) \mapsto v : (n, p) \in F \oplus L]$ is the same as $av$ except it maps $(p, x)$ to $v$ for each $p$ such that $(n, p) \in F \oplus L$.

Definition 4.9 (Update function of econfs). We define function $\text{udec} : N \times \text{State} \rightarrow \text{Econf}$ as follows:

\[\text{udec}(n, (av, ec)) = ec',\]

where $ec'$ is determined according to $n$, $av$ and $ec$ as follows.

- $n$ is any type.
  - $ec'(c(p, n)) := \text{unchk}$, if $c(p, n) \in C$.
  - $ec'(l(p, n)) := \text{chk}$, if $l(p, n) \in L$.
- $n = (x := e)$.
  - $ec'((n, p)) := \text{chk}$, if $(n, p) \in F \oplus D$.
- $n = (\text{if } e)$ and $E(n, e)av = T$.
  - $ec'(ct(n, p)) := \text{act}$, if $ct(n, p) \in C$.
  - $ec'((q, r)) := \text{unchk}$, if $q \in G_T(n) - \{n\}$ and $(q, r) \in F \oplus D$.
  - $ec'(l(r, q)) := \text{unchk}$, if $q \in G_T(n) - \{n\}$, $r \in G_T(n)$ and $l(r, q) \in L$.
  - $ec'((q, r)) := \text{chk}$, if $q \in G_F(n) - G_T(n)$ and $(q, r) \in F \oplus D$. 

12
\[
\begin{align*}
- \quad ec'(l(q, r)) & := \text{unchk, if } q \in G_F(n) - G_T(n) \text{ and } \, l(q, r) \in L. \\
- \quad ec'(l(r, q)) & := \text{chk, if } q \in G_F(n) - G_T(n) \text{ and } r \notin G_F(n) - G_T(n) \text{ and } \, l(r, q) \in L.
\end{align*}
\]

- \[ n = (\text{if e}) \text{ and } E(n, e)av = F \] This case is the same as the case \[ E(n, e)av = T. \] Replace ct with cf and \[ G_T(n) \text{ with } G_F(n) \] in the definition.

Def. 4.10 (Operational semantics of the PDG). Let \( G = (N, C, F, L, D) \) be a PDG. A run of \( G \) is a sequence of states \( s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \ldots \) where \( s_i \in \text{State} \) and \( n_i \in N \). Let \( s_i = (av_i, ec_i) \), \( av_0 \) be a given initial avail and \( ec_0 \) be the initial econf defined as follows:

\[
\forall (p, q) \in C. ec_0((p, q)) = \begin{cases} 
\text{act} & \text{if } p = \text{entry} \\
\text{unchk} & \text{otherwise}
\end{cases},
\]

\[
\forall (p, q) \in F \oplus L \oplus D. ec_0((p, q)) = \text{unchk}
\]

Then \( n_i \) and \( s_{i+1} \) are determined as follows:

\[
\begin{align*}
\quad s_i \xrightarrow{n_i} s_{i+1} & \defeq n_i \in \text{Next}(s_i) \land \\
\quad & \text{av}' = udav(n_i, av_i) \land ec' = udec(n_i, s_i).
\end{align*}
\]

Then a PDG is either finite or infinite. If a run is finite and the last state is \( s \), then \( \text{Next}(s) = \emptyset \).

4.2 Deterministic PDG

A PDG does not necessarily have a unique run even if it starts with the same initial state, because it can express programs more expressive than usual sequential ones. However, we can prove that for a specific class of PDGs, called deterministic PDGs (dPDGs), it has runs that end with the same state from the same initial state, if they are finite. In this section, we define the class of dPDGs and prove some properties of it.

We first introduce the notion of the minimal common ancestor (mca).

Def. 4.11 (Minimal common ancestors). Let \( G = (N, C, F, L, D) \) be a PDG and \( p, q \in N \). The minimal common ancestors of nodes \( p \) and \( q \) are the last nodes of the longest common prefix of the acyclic paths from entry to \( p \) and \( q \) in \( (N, C) \). We write \( mca(p, q) \) for the set of minimal common ancestors of \( p \) and \( q \).

Def. 4.12 (Deterministic PDGs). A PDG \( G = (N, C, F, L, D) \) is a deterministic PDG if it satisfies the following three conditions where the term \( LP \) denotes the set of loops in the CDG \( (N, C) \):

1. ...
1. \((p, n) \in C \wedge (q, n) \in C\) implies \(p \in G^*(q)\) or \(q \in G^*(p)\) or \(\forall r \in mca(p, q). \forall Q \in \{T, F\}. \{p, q\} \not\subseteq G^*_Q(r)\).

2. \(f_x(p, u) \in F \wedge f_x(q, u) \in F \wedge \exists r \in mca(p, q). \exists Q \in \{T, F\}. \{p, q\} \subseteq G^*_Q(r)\) implies \(d(p, q) \in D \lor d(q, p) \in DD\)

3. \((p, q) \in F \oplus D\) implies \(\forall R \in LP. \forall r \in R, p \in G(r) \Rightarrow q \in G^*(r)D\)

Condition 1 states that if a node \(n\) is control-dependent on both \(p\) and \(q\), then \(p\) and \(q\) cannot be executable simultaneously. Fig. 9 illustrates this condition. The outgoing edges from any minimal common ancestor \(r\) of \(p\) and \(q\) must have different labels between the one that leads to \(p\) and the one to \(q\). Otherwise either \(p\) must be controlled by \(q\) or \(q\) must be controlled by \(p\).

Condition 2 states that if \(p\) and \(q\) define the same variable \(x\), have a common \(F\)-successor and can be executable simultaneously, then there must be a def-order dependence between \(p\) and \(q\). This is illustrated in Fig. 10. In this case, \(p\) and \(q\) can be simultaneously executable. If there is no def-order dependence between \(p\) and \(q\), the value of the variable at node \(n\) can be different depending on which node (i.e., \(p\) or \(q\)) is executed first.

Condition 3 states that if there is data dependence from \(p\) to \(q\) and \(p\) is control-dependent on a node \(r\) in some loop \(R\), then \(q\) must be control-dependent
Figure 11: Condition 3 of dPDG

Figure 12: An example CFG whose corresponding PDG is not dPDG.

on some node in that loop. Fig. 11 depicts the condition, where \( r' \) corresponds to 'some node' in the previous sentence (note that \( r' \in G^+(r) \) because both are contained in the same loop). In Fig. 11, if node \( q \) is not contained in \( G(r') \) then whether \( q \) satisfies \( p_{condC} \) can be determined independently on a condition of if statements in \( R \). Meanwhile, since \( p \) is contained in \( G(r) \), executability of \( p \) depends on \( r \). Since \( R \) is a loop, \( r \) can be repeated, which implies that \( p \) can be repeated many times and the value of a variable used in \( q \) may be changed many times. However, as stated above, \( q \) can satisfy \( condC \) independently on the loop. Therefore, once \( p \) is executed and data dependence to \( q \) is satisfied, \( q \) can be executable. Then the result of execution of the PDG depends on how many times \( p \) is repeated when \( q \) is executed.

Actually, there are PDGs constructed from CFGs that are not dPDGs. CFGs like the one in Fig. 12 do not satisfy condition 1 of dPDGs, where a back edge from some if statement comes into a branch.

As for Condition 2 and 3, we can prove that for all CFGs, their PDGs satisfy them.
Theorem 4.1. A PDG constructed from a CFG satisfies Condition 2 and 3 of Definition 4.12.

Proof. We prove the claim separately.

Condition 2 In the corresponding PDG, suppose that for statements $p, q$ and $u$ and variable $x$, we have $f_x(p, u) \in F \land f_x(q, u) \in F \land \exists r \in mca(p, q). \exists Q \in \{T, F\}. \{p, q\} \subseteq G^*(r)$. By the definition of loop-independent data dependence, $u$ is reachable from both $p$ and $q$ in the CFG. By the definition of control dependence, both $p$ and $q$ are reachable from a $T$-successor of $r$ or a $F$-successor of $r$. Since $r$ is a mca of $p$ and $q$, $p$ is reachable from $q$ or $q$ is reachable from $p$ in the CFG. Then, by the definition of def-order dependence, we have $\text{DefOrd}(p, q)$ or $\text{DefOrd}(q, p)$.

Condition 3 In the corresponding PDG, suppose that for statements $p$ and $q$, we have $(p, q) \in F \oplus D$, and for a node $r$ in some loop $R$ in the CDG, we have $p \in G(r)$. By Lemma 4.1 (below), a node in a loop in the CDG is an iteration statement in a loop in the corresponding CFG. Therefore, $r$ is an iteration statement in the CFG. Since $(p, q) \in F \oplus D$, $q$ is reachable from $p$ in the CFG. Moreover, since $p \in G(r)$, $p$ is reachable from $r$ in the CFG. Thus, $q$ is also reachable from $r$ in the CFG. Since $r$ is an iteration statement of some loop in the CFG, by Lemma 4.2 (below) we have $q \in G^*(r)$.

Lemma 4.1. Let $P$ be a CFG and $G$ be the CDG of the corresponding PDG. A node in a loop in $G$ is an iteration statement in a loop in $P$.

Proof. Let $p$ and $q$ be nodes in a loop $R'$ in $G$. By the definition of the loop, $p$ is reachable from $q$ and vice versa in $G$. Since $p$ and $q$ have outgoing $C$-edges, they are if statements. By the definition of control dependence, if a node is reachable from another node in $G$, it is also reachable in $P$. Thus, $p$ and $q$ are if statements in some loop $R$ in $P$. In other words, we showed that a node in a loop in $G$ is contained in some loop in $P$. Then there exists at least one loop iteration statement in $R$ by Definition 4.2. We call it $r$. Let $s \in R'$ be a statement such that $(s, r) \in C$. By the definition of control dependence, $s$ is not post-dominated by $r$ in $P$, which implies $s$ has an outgoing edge that comes into a node inside $R$ and one that comes into a node outside $R$. Thus, by the definition of the iteration statement, $s$ is an iteration statement. Since $s$ is an iteration statement, any node $s' \in R'$ such that $(s', s) \in C$ is again an iteration statement by the above argument. Thus, we proved that every node in a loop in $G$ is an iteration statement.

Lemma 4.2. Let $P$ be a CFG and $G$ be the CDG of the corresponding PDG. If $p$ is a statement reachable from some node in a loop $R$ in $P$, then for any iteration statement $r \in R$, we have $p \in G^*(r)$.
Proof. If \( p \) is a statement in \( R \), we have \( p \in G(r') \) for some iteration statement \( r' \) of \( R \). If \( p \) is not a statement in \( R \), by the definition of control dependence, \( p \in G^*(r') \) for some iteration statement \( r' \) of \( R \). Here, by Lemma 4.1, any iteration statement \( r \) contained in \( R \) is contained in a loop in \( G \) that contains \( r' \). Therefore, we have \( r' \in G^*(r) \) and \( p \in G^*(r) \) holds.

5 Semantical correspondence between CFG and PDG

In this section, we prove the semantical correspondence between a CFG and its corresponding PDG. Here the important assumption is that the PDG should be a dPDG. Hence, our claim is that if the PDG constructed from a CFG is a dPDG, then the results of the executions of the CFG and the PDG coincide. This is derived from the following two properties:

Property 1 There is the same execution order of statements for both a CFG and its corresponding PDG (Theorem 5.1).

Property 2 In a dPDG, every run starting with the same initial state has the same final state (Theorem 5.2).

By these properties, if a PDG corresponding to a CFG is a dPDG, the execution of the CFG and any execution of the PDG have the same final result.

To prove these theorems, we need a number of lemmata. To make the presentation easy and understandable, we introduce some notations. Hereafter, we write \( P = (N, E) \) for a CFG, \( G = (N \cup \{\text{entry}\}, C, F, L, D) \) for the corresponding PDG, \( \sigma \) or \( \sigma_i \) for stores and \( s = (av, ec) \) or \( s_i = (av_i, ec_i) \) for states. If \( Q \in \{T, F\} \), we write \( \overline{Q} \) for the negation of \( Q \). We write \( \left[i, j\right] \) for the set of integers from \( i \) to \( j \) and \( (i, j) \) for the set of integers from \( i + 1 \) to \( j - 1 \).

We define a relation that represents a state (i.e., store) of a run of a CFG is ‘equal’ to a state of a run of a PDG. Intuitively, it says that a variable used at node \( n \) is the same in \( \sigma \) and \( av \).

Definition 5.1. Let \( \sigma \) be a store and \( av \) be an avail. Let \( N \) be a set of nodes. For \( n \in N \), we write \( \text{use}(n) \) for the set of variables used in \( n \). We define a binary relation \( \sigma \approx_n av \) as follows:

\[
\forall x \in \text{use}(n).\sigma(x) = av(n, x).
\]

We simply write \( \sigma \approx av \) for \( \forall n \in N.\sigma \approx_n av \).

Next, we define a notation that represent reachable and unreachable nodes from some node in a CFG.
Definition 5.2. Let $n \in \mathbb{N}$. We define the following sets:

$$UR(n) = \{ m \mid m \text{ is unreachable from } n \text{ in } P \}$$
$$UR'(n) = \{ m \mid m \text{ is unreachable from } n \text{ in the graph obtained by removing all back edges from } P \}$$
$$R(n) = \{ m \mid m \text{ is reachable from } n \text{ in } P \}$$
$$R'(n) = \{ m \mid m \text{ is reachable from } n \text{ in the graph obtained by removing all back edges from } P \}$$

5.1 Proof of Property 1

The fact that a PDG can execute sentences in the same order as the corresponding CFG is executed means that any statement that should be executed next in the CFG is also executable at the current state of the run of the PDG, that is, it satisfies $condCFLD$ (Lemma 5.7). Therefore our first target is to prove this fact.

When a statement is to be executed next in a CFG run, a statement on which it is loop-independently data-dependent or def-order-dependent should have already been executed, or is contained in the branch of some conditional statement that are not executed. The next two lemmata formally describe this fact. They states that if node $q$ is loop-independently data-dependent or def-order-dependent on $p$, $p$ is not reachable from $q$ in the graph obtained by removing all back edges from the CFG.

Lemma 5.1. For all $p, q \in \mathbb{N}$, if $LIDD(p, q)$ holds then $p \in UR(q)$ or $p \in UR'(q) - UR(q)$ holds.

Proof. Since $p \in UR(q)$ or $p \in UR'(q) - UR(q)$ implies $p \in UR(q) \cup UR'(q)$ and $UR(q) \subseteq UR'(q)$, we just need to show $p \in UR'(q)$. Suppose that $LIDD(p, q)$ and $p \notin UR'(q)$ hold. $p \notin UR'(q)$ is equivalent to $p \notin R'(q)$. Thus, $p$ is reachable from $q$ without passing any back edge. Since $LIDD(p, q)$ holds, by the definition of loop-independent data dependence, either i) $q$ is reachable from $p$ and $p$ and $q$ are not contained in the same loop, or ii) if $p$ and $q$ are contained in the same loop then there is a reaching path from $q$ to $p$ that does not contain any back edge. However, both cases contradict to $p \notin R'(q)$.

Lemma 5.2. For all $p, q \in \mathbb{N}$, if $DefOrd(p, q)$ holds then $p \in UR(q)$ or $p \in UR'(q) - UR(q)$ holds.

Proof. It is again sufficient to show $p \in UR'(q)$. Suppose $DefOrd(p, q)$ and $p \notin UR'(q)$. $p \notin UR'(q)$ is equivalent to $p \notin R'(q)$. Thus, $p$ is reachable from $q$ without passing any back edge. Since $DefOrd(p, q)$ holds, by the definition of def-order dependence, either i) $q$ is reachable from $p$ and $p$ and $q$ are not contained in the same loop, ii) $p$ and $q$ are contained in the same loop and $q$ is reachable from $p$ without passing any back edge, or iii) $p$ and $q$ are contained in the same loop, $p$ is reachable from $q$ and vice versa, but only by passing back edges. However, either case contradicts to $p \notin R'(q)$.
The next lemma states that if a run of a PDG corresponds to a run of the corresponding CFG, then at the current state of the run of the PDG, any loop-independent data dependence and def-order dependence from the statements that are already executed in a run of the CFG are satisfied (i.e., the states of the edges are $\text{chk}$).

**Lemma 5.3.** Suppose that there is a run $n_0 \rightarrow s_1 \rightarrow \cdots \rightarrow n_k \rightarrow s_k$ of $G$ that corresponds to a run $n_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow n_k \rightarrow \sigma_k$ of $P$, that is to say, $\forall i. \sigma_i \approx_{n_i} \text{av}_i$. If $\sigma_k \rightarrow \sigma_{k+1}$, the following holds:

$$\forall p \in \text{UR}'(n_k).(p,q) \in F \oplus D \Rightarrow \text{ec}_k(p,q) = \text{chk}.$$  

**Proof.** Since $\text{UR}(n) \subseteq \text{UR}'(n)$, the target of the proof can be divided into the following two claims.

1. If $n_{k-1} = (x := e)$, $n_k$ has a unique successor.

   (a) If $n_k$ and $n_{k-1}$ are not contained in the same loop, we have $\text{UR}(n_k) = \text{UR}(n_{k-1}) \cup \{n_{k-1}\}$. By the induction hypothesis, we have $\forall p \in \text{UR}(n_{k-1}) \forall (p,q) \in F \oplus D. \text{ec}_{k-1}(p,q) = \text{chk}$. By the definition of $\text{udec}$, $\text{ec}_k(n_{k-1},q) = \text{chk}$ and $\forall (p,q) \in F \oplus D. \exists (p,q) \in n_{k-1} \Rightarrow \text{ec}_k(p,q) = \text{ec}_{k-1}(p,q)$. Therefore, claim (1) holds. Since $\text{UR}'(n_k) = \text{UR}(n_k)$, claim (2) also holds.

   (b) If $n_k$ and $n_{k-1}$ are contained in the same loop, $\text{UR}(n_k) = \text{UR}(n_{k-1})$. Since $n_{k-1} \notin \text{UR}(n_k)$, by the induction hypothesis and the definition of $\text{udec}$, we have $\forall p \in \text{UR}(n_k) \forall (p,q) \in F \oplus D \Rightarrow \text{ec}_k(p,q) = \text{chk}$. That is, claim (1) holds. Meanwhile, because $\text{UR}'(n_k) = \text{UR}'(n_{k-1}) \cup \{n_{k-1}\}$,

\[
\text{UR}'(n_k) - \text{UR}(n_k) = (\text{UR}'(n_{k-1}) \cup \{n_{k-1}\}) - \text{UR}(n_{k-1}) \\
= (\text{UR}'(n_{k-1}) - \text{UR}(n_{k-1})) \cup \{n_{k-1}\} - \text{UR}(n_{k-1}) \\
= (\text{UR}'(n_{k-1}) - \text{UR}(n_{k-1})) \cup \{n_{k-1}\} \quad (\because n_{k-1} \notin \text{UR}(n_{k-1}))
\]

By the induction hypothesis and the definition of $\text{udec}$, we have $\forall p \in \text{UR}'(n_{k-1}) \rightarrow \text{UR}(n_{k-1}).(p,q) \in F \oplus D \Rightarrow \text{ec}_k(p,q) = \text{chk}$. Again by the definition of $\text{udec}$, $\forall (n_{k-1},q) \in F \oplus D. \forall (p,q) \in \text{UR}(n_{k-1}) \Rightarrow \text{ec}_k(p,q) = \text{chk}$. This implies claim (2).

2. If $n_{k-1} = (\text{if} e)$, assume that $\mathcal{E}(e)\sigma_{k-1} = \mathcal{E}_{av}(n_{k-1},e)\text{av}_{k-1} = \mathbf{T}$. Suppose $(n_{k-1},p_t) \in E$ and $(n_{k-1},p_f) \in E$. Note that $n_k = p_t$. 

19
Therefore, there is a statement whose label is \( Q \) from \( p \) reachable from \( q \). Thus, by the definition of control dependence. Therefore, claim (1) holds as in case 1-(b). Meanwhile, we have \( n_{k-1} \) and \( n_{k-1} \) contained in the same loop, \( UR(n_{k-1}) = UR(n_{k-1}) \). Therefore, claim (1) holds as in case 1-(a). Meanwhile, we have \( UR'(n_{k-1}) = UR'(n_{k-1}) \). Therefore, claim (2) holds.

The above lemma will be used to prove that when a PDG is executed in the same order as the corresponding CFG is executed, the statements that should be executed next in the CFG satisfy \( \text{condF} \) and \( \text{condD} \) at the current state of the run of the PDG.

**Lemma 5.4.** Suppose that \( p \) is an if statement and has outgoing looping edges whose label is \( Q \). We assume \( \exists r \neq p, \exists p' \in G_Q(r). \{p, p'\} \subset G_Q(r) \). If node \( q \) is reachable from \( p \) only via a looping edge outgoing from \( p \), then \( q \) is reachable from \( p \) only via a back edge in the CFG \( P \).

**Proof.** Since \( Q \)-edges from \( p \) in \( G \) are looping edges, \( G_Q(p) \) consists of a loop in \( P \). Therefore, there is a \( q' \) in the loop that has an incoming edge from the outside of the loop. The reason why such \( q' \) exists is because there is \( r \neq p \) such that \( q' \in G_Q(r) \). By the definition of the back edge, the edge \( (p, q')_Q \in E \) of \( P \) is a back edge of the loop. Now assume that a node \( q \) in the loop is reachable from \( p \) only via a looping edge outgoing from \( p \). That means \( q \) is reachable from \( q' \) by the definition of control dependence. Therefore, \( q \) is reachable from \( p \) only via the back edge \( (p, q')_Q \).
The next lemma states that if there exists a run of $G$ corresponding to the execution of $P$, the states of the $C$-edges outgoing from the statements in the subgraph of the statement that will be executed next in $P$ are not act.

**Lemma 5.5.** Suppose that there is a run $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_k$ of $G$ that corresponds to a run $\sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_k$ of $P$, that is to say, $\forall i. \sigma_i \approx_{n_i} \text{av}_i$. If $\sigma_k \rightarrow \sigma_{k+1}$, then $\forall p \in G^*(n_k). p \neq n_k \Rightarrow \forall c(p,q) \in C. e_{ck}(c(p,q)) \neq \text{act}$.

**Proof.** By induction on $k$.

**Base Case** $k = 0$. By the definition of the operational semantics of the PDG, we have $\forall(p,q) \in C. e_{c0}(p,q) = \text{act} \Rightarrow p = \text{entry}$. If $n_0$ is not an $\text{if}$ statement, the claim immediately follows. If $n_0$ is an $\text{if}$ statement, by the definition of the augmented CFG, it is not the case that $(\text{entry}, q) \in C$ for some $q \in G^*(n_0)$. Thus, the claim follows.

**Induction Step** Since the claim trivially follows if $n_k$ is not an $\text{if}$ statement, we only consider the case where $n_k$ is an $\text{if}$ statement. For some $i < k$, we consider the case where there is $n_i$ such that $CD(n_i, n_k) \land \forall j \in (i, k). \lnot CD(n_j, n_k)$. For $p \in G^*(n_k)$, since $G^*(n_k) \subseteq G^*(n_i)$, by the induction hypothesis, $e_{ci}(c(p,q)) \neq \text{act}$. Since node $n_j$ for any $j \in (i, k)$ does not have a $C$-edge pointing to a node in $G^*(n_k)$, we have $e_{ck}(c(p,q)) = e_{ci}(c(p,q)) \neq \text{chk}$ by the definition of $udec$.

The next lemma states that if there exists a run of $G$ corresponding to the execution of $P$, the loop-carried data dependence from the statement that will be executed next in $P$ is satisfied.

**Lemma 5.6.** Suppose that there is a run $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_k$ of $G$ that corresponds to a run $\sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_k$ of $P$, that is to say, $\forall i. \sigma_i \approx_{n_i} \text{av}_i$. If $\sigma_k \rightarrow \sigma_{k+1}$, then $\forall l(n_k, q) \in L. e_{ck}(l(n_k, q)) = \text{chk}$.

**Proof.** If $(n_k, q) \in L$, by the definition of the loop-carried data dependence, $n_k$ and $q$ reside in the same loop and every reaching path from $n_k$ to $q$ passes a back edge of the loop. Suppose that $p \in mca(n_k, q)$ be the last executed minimal common ancestor of $n_k$ and $q$. If $q$ was executed during the execution from $p$ to $n_k$, the state of $l(n_k, q)$ turned to be $\text{chk}$ according to $udec$. Let $o$ be an arbitrary $\text{if}$ statement executed after $q$’s execution. Then we have $\forall Q \in \{T, F\}. \{q, n_k\} \not\in G_Q(o)$. Otherwise, $o$ should have been the last executed minimal common ancestor of $q$ and $n_k$. Therefore, if $n_k \not\in G(o)$, the state of $l(n_k, o)$ will not be changed by the execution of $o$. Meanwhile, if we assume $n_k \in G_T(o)$ (the case of $G_F(o)$ is the same), since $n_k$ will be executed, the condition of $o$ was evaluated to $T$. Thus, by the definition of $udec$, the state of $l(n_k, q)$ will not be changed. As a consequence, we have $e_{ck}(l(n_k, q)) = \text{chk}$. If $q$ is not executed during the execution from $p$ to $n_k$, there is an $\text{if}$ statement $o$ such that $q \in G(o)$ (note that $o$ may coincide with $p$). If we assume $q \in G_T(o)$ (the case of $G_F(o)$ is the same), since $q$ was not executed, the condition of $o$ had been evaluated to $F$ and $q \in G_F(o) - G_T(o)$. By the same argument as above, we have $\forall Q \in \{T, F\}. \{q, n_k\} \not\in G_Q(o)$. By the definition of $udec$, the state of
Proof of cond$L$

Proof of cond$F$

Proof of cond$C$

$\sigma$ corresponds to a run of the CFG satisfies cond$L$ in the current state of the run of the PDG.

By using the lemmata 5.3 and 5.6, we can show that there is a run of $G$ that has the same execution order as that of $P$.

**Lemma 5.7.** Suppose that there is a run $s_0 \rightarrow s_1 \rightarrow \cdots \rightarrow s_k$ of $G$ that corresponds to a run $\sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_k$ of $P$, that is to say, \( \forall i, \sigma_i \approx_n a v_i \).

If $\sigma_k \rightarrow q, n_k \in \text{Next}(s_k)$. Then, by Lemma 5.6, we have $ec_k(c(q, n_k)) = act$. Since the execution of $\sigma_k$ does not change the state of $c(q, n_k)$ by the definition of $udec$, we have $ec_k(c(q, n_k)) = act$. If $n_{k-1}$ is an if statement, we have $c(n_{k-1}, n_k) \in C$. Since $\sigma_k \rightarrow q, n_k \in \text{Next}(s_k)$, we have $ec_k(c(n_k, p)) = act$ by the definition of $udec$. Thus, conjunct (3) follows.

By Lemma 5.6 conjunct (4) follows.

**Proof of cond$F(s_k, n_k)$**. For all $f(p, n_k) \in F$, by Lemma 5.4 $p \in UR(n_k)$ or $p \in UR(n_k) - UR(n_k)$ holds. Then, by Lemma 5.4 we have $ec_k(f(p, n_k)) = chk$ and the claim follows.

**Proof of pcond$D(s_k, n_k)$**. For all $d(p, n_k) \in F$, by Lemma 5.2 $p \in UR(n_k)$ or $p \in UR(n_k) - UR(n_k)$ holds. Then, by Lemma 5.3 we have $ec_k(f(p, n_k)) = chk$ and the claim follows.

**Proof of cond$L(s_k, n_k)$**. It immediately follows from Lemma 5.6

---

By this lemma, the following important theorem is proved. This theorem states that for any run of $P$ there is a corresponding run of $G$.

---

1Here we assume that $n_{k-1}$ has successors $n_k$ and the one different from $n_k$. Otherwise the above argument applies.
**Theorem 5.1.** Suppose $\sigma_0 \approx av_0$. For a run $\sigma_0 \xrightarrow{n_0} \sigma_1 \xrightarrow{n_1} \cdots \xrightarrow{n_k} \sigma_{k+1}$ of $P$, there is a run $s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \cdots \xrightarrow{n_k} s_{k+1}$ of $G$ that satisfies $\forall i \in [0, k]. \sigma_i \approx_{n_i} av_i$.

**Proof.** By induction on $k$.

**Base Case** $k = 0$. For a run $\sigma_0 \xrightarrow{n_0} \sigma_1$ of $P$, since we know $\sigma_0 \approx av_0$ by the assumption, $n_0 \in Next(s_0)$ follows by Lemma 5.7. Henceforth, there is a run $s_0 \xrightarrow{n_0} s_1$ of $G$. Moreover, by the assumption we have $\sigma_0 \approx_{n_0} av_0$.

**Induction Step** By the induction hypothesis, for $P$’s run $\sigma_0 \xrightarrow{n_0} \sigma_1 \xrightarrow{n_1} \cdots \xrightarrow{n_k} \sigma_{k+1}$, there is a $G$’s run $s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \cdots \xrightarrow{n_k} s_{k+1}$ that satisfies $\forall i \in [0, k]. \sigma_i \approx_{n_i} av_i$. Suppose that $\sigma_{k+1} \xrightarrow{n_{k+1}} \sigma_{k+2}$. By Lemma 5.7 we have $s_{k+1} \in Next(s_{k+1})$. Thus, there is a $G$’s run $s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \cdots \xrightarrow{n_k} s_{k+1}$.

We then show $\sigma_{k+1} \approx_{n_{k+1}} av_{k+1}$, that is to say, $\forall x \in use(n_{k+1}). \sigma_{k+1} = av_{k+1}(n_{k+1}, x)$. For $x \in use(n_{k+1})$, if an assignment statement defining $x$ is executed between $s_0$ and $s_k$, then let the last one among such statements be $n_j = (x := e)$. Then we have $(n_j, n_{k+1}) \in F \div L$. By the definition of $udav$, $av_{k+1}(n_{k+1}, x) = \mathcal{E}(n_j, e)av_j$. Since we have $\sigma_j \approx_{n_j} av_j$ by the induction hypothesis, $\mathcal{E}(e)\sigma_j = \mathcal{E}(n_j, e)av_j$ follows. Meanwhile, by the operational semantics of the CFG, we have $\sigma_{j+1}(x) = \mathcal{E}(e)\sigma_j$. Hence, $\sigma_{j+1}(x) = av_{j+1}(n_{j+1}, x)$. Since no assignment statements defining $x$ are executed between $n_j$ and $n_{k+1}$, we have $\sigma_{j+1}(x) = \sigma_{k+1}(x)$ and $av_{j+1}(n_{j+1}, x) = av_{k+1}(n_{k+1}, x)$. Therefore, $\sigma_{k+1}(x) = av_{k+1}(n_{k+1}, x)$ follows. If no assignment statements defining $x$ are executed between $n_0$ and $n_k$, then we have $\sigma_{k+1}(x) = \sigma_0(x)$ and $av_{k+1}(n_{k+1}, x) = av_0(n_{k+1}, x)$. Since $\sigma_0 \approx av_0$ from the assumption, $\sigma_{k+1}(x) = av_{k+1}(n_{k+1}, x)$ follows. $\blacksquare$

### 5.2 Proof of Property 2

Next we prove Property 2. Property 2 says that all runs of a dPDG starting with the same initial state end with the same final state (if they terminate). This property is derived from confluence of the runs of a dPDG (Lemma 5.17) which says if there are multiple executable statements in a run of a dPDG, the order of execution does not change the final result. This property is proved by the fact that an execution of a statement does not affect executability and the execution results of the other statements that are simultaneously executable in the dPDG (Lemma 5.10, 5.11, 5.13, and 5.15). Other lemmata are used to prove the above lemmata or Theorem 5.2.

The next lemma states that two executable nodes at a run of a PDG are contained in the subgraph of the same truth value of an `if` statement.

**Lemma 5.8.** Let $s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \ldots$ be a run of a PDG. Suppose $c(o_1, p) \in C$, $c(o_2, q) \in C$, $p \notin G^*(q)$ and $q \notin G^*(p)$. For any $i$, if $ec_i(o_1, p) = ec_i(o_2, q) = act$ then $\exists r \in mca(p, q). \exists Q \in \{T, F\}. \{p, q\} \subseteq G^*_Q(r)$.

**Proof.** By induction on $i$.

**Base Case** If $i = 0$, $o_1 = o_2 = entry$. Obviously, $\{p, q\} \subseteq G_T(entry)$. Since $mca(p, q) = \{entry\}$, the claim holds.

**Induction Step**
1. If \( ec_{i-1}(o_1, p) = ec_{i-1}(o_2, q) = \text{act} \), by the induction hypothesis the claim holds.

2. If \( ec_{i-1}(o_1, p) \neq \text{act} \) and \( ec_{i-1}(o_2, p) \neq \text{act} \), then by the definition of \( udec \) and \( \text{condC} \), \( o_1 = o_2 = n_{i-1} \) and \( \exists Q \in \{T, F\}. \{p, q\} \subseteq G^*_Q(n_{i-1}) \) must be the case. Here suppose \( n_{i-1} \notin \text{mca}(p, q) \), then in the CDG either \( p \) or \( q \) must lie on a path from \( \text{entry} \) to \( n_{i-1} \). This contradicts to either \( q \in G^*(p) \) or \( p \in G^*(q) \). Henceforth, \( n_{i-1} \in \text{mca}(p, q) \).

3. If \( ec_{i-1}(o_1, p) = \text{act} \) and \( ec_{i-1}(o_2, p) \neq \text{act} \), then \( n_{i-1} = o_2 \) and must be some \((o, o_2) \in C \) such that \( ec_{i-1}(o, o_2) = \text{act} \) by the definition of \( udec \) and \( \text{condC} \). By the induction hypothesis, there is \( r_1 \in \text{mca}(p, o_2) \) and \( Q \in \{T, F\} \) such that \( \{p, o_2\} \subseteq G^*_Q(r_1) \). Since \( \{o_2, q\} \in C \), we have \( \{p, q\} \subseteq G^*_Q(r_1) \). If we assume that \( r_1 \notin \text{mca}(p, q) \), either \( p \) or \( q \) must lie on a path from \( \text{entry} \) to \( r_1 \) in the CDG. This contradicts to either \( q \in G^*(p) \) or \( p \in G^*(q) \). Henceforth, \( n_{i-1} \in \text{mca}(p, q) \).

\[\Box\]

**Lemma 5.9.** Let \( s_0 \xrightarrow{o_1} s_1 \xrightarrow{o_2} \ldots \) be a run of a PDG. For any \( p \) and \( q \) \( (p \neq q) \), if \( \{p, q\} \subseteq \text{Next}(s_i) \), then \( p \notin G^*(q) \) and \( q \notin G^*(p) \).

**Proof.** \( p \in G^*(q) \) contradicts to the second conjunct of the predicate \( \text{condC} \). \( q \in G^*(p) \) does the same. \[\Box\]

**Lemma 5.10.** Let \( s_0 \xrightarrow{o_1} s_1 \xrightarrow{o_2} \ldots \) be a run of a PDG. For any \( p \) and \( q \) \( (p \neq q) \), if \( \{p, q\} \subseteq \text{Next}(s_i) \) then there are no \( F^- \) and \( L \)-edges between \( p \) and \( q \).

**Proof.** We split the proof into the following cases.

- **Let** \((p, q) \in F \). Since \( p \in \text{Next}(s_i) \), there is \((o, p) \in C \) such that \( ec_i(o, p) = \text{act} \). Therefore, there is some \( j < i \) such that \( n_j = o \) and \( \forall k \in (j, i). ec_k(o, p) = \text{act} \) (this implies \( p \) is not executed between \( s_j \) and \( s_i \)). By the definition of \( udec \), \( ec_{j+1}(f(p, q)) = \text{unchk} \). Furthermore, by the definition of \( \text{condC} \), we have \( \forall k \in (j, i). p \notin G(n_k) \) since \( ec_k(o, p) = \text{act} \). Therefore, \( ec_i(f(p, q)) = ec_{j+1}(f(p, q)) = \text{unchk} \). This contradicts to \( q \in \text{Next}(s_i) \).

- **Let** \((p, q) \in L \). Since \( q \in \text{Next}(s_i) \), there is \((o, q) \in C \) such that \( ec_i(o, q) = \text{act} \). Therefore, there is some \( j < i \) such that \( n_j = o \) and \( \forall k \in (j, i). ec_k(o, q) = \text{act} \) (this implies \( q \) is not executed between \( s_j \) and \( s_i \)). By the definition of \( udec \), \( ec_{j+1}(l(p, q)) = \text{unchk} \). Furthermore, by the definition of \( \text{condC} \), we have \( \forall k \in (j, i). q \notin G(n_k) \) since \( ec_k(o, q) = \text{act} \). Therefore, \( ec_i(l(p, q)) = ec_{j+1}(l(p, q)) = \text{unchk} \). This contradicts to \( p \in \text{Next}(s_i) \).

- The proofs of the cases for \((q, p) \in F \) and \((q, p) \in L \) are the same as above. \[\Box\]

**Lemma 5.11.** Let \( s_0 \xrightarrow{o_1} s_1 \xrightarrow{o_2} \ldots \) be a run of a deterministic PDG. For any \( p \) and \( q \) \( (p \neq q) \), if \( f_x(p, n) \in F \) and \( f_x(q, n) \in F \) for some \( n \), then \( \{p, q\} \notin \text{Next}(s_i) \) holds.
Proof. Suppose \( \{p, q\} \subseteq \text{Next}(s_i) \). Since \( p \) and \( q \) are assignment statements, we know \( p \notin G^*(q) \) and \( q \notin G^*(p) \). By Lemma 5.8, there must be some \( r \in \text{mca}(p, q) \) and \( Q \in \{T, F\} \) such that \( \{p, q\} \subseteq G^*_Q(r) \). Then, by the condition of the dPGD, either \( d(p, q) \in D \) or \( d(q, p) \in D \) holds. We consider the case \( d(p, q) \in D \). Since \( p \in \text{Next}(s_i) \), there is \((o, p) \in C \) such that \( ec_i(o, p) = \text{act} \). Therefore, there is some \( j < i \) such that \( n_j = o \) and \( \forall k \in (j, i). ec_k(o, p) = \text{act} \) (this implies \( p \) is not executed between \( s_j \) and \( s_i \)). By the definition of \( udec \), \( ec_{j+1}(d(p, q)) = \text{unchk} \). Furthermore, by the definition of \( \text{condC} \), we have \( \forall k \in (j, i). p \notin G(n_k) \) since \( ec_k(o, p) = \text{act} \). Therefore, \( ec_i(d(p, q)) = ec_{j+1}(d(p, q)) = \text{unchk} \). This contradicts to \( q \in \text{Next}(s_i) \).

**Lemma 5.12.** If \( o \in G^*(p) \) and \( p \notin G^*(q) \) and \( q \notin G^*(p) \), then \( \text{mca}(p, q) \subseteq \text{mca}(o, q) \).

Proof. Since CDGs are connected and \( \text{entry} \) is the root, there is a non-cyclic path \( \pi = \text{entry} \ldots r \) in the CDG. Suppose \( r \in \text{mca}(p, q) \). By the definition of the mca, we have two non-cyclic paths \( \pi_1 = \pi \ldots p \) and \( \pi_2 = \pi \ldots q \). Since \( o \in G^*(p) \), there is a non-cyclic path \( \pi_3 = \pi \ldots o \). Here since the longest common prefix of \( \pi_1 \) and \( \pi_2 \) is \( \pi \), and \( q \notin G^*(p) \) and \( p \notin G^*(q) \), we have \( r \neq p \) and \( r \neq q \). Therefore, the longest common prefix of \( \pi_2 \) and \( \pi_3 \) is \( \pi \). This shows \( r \in \text{mca}(o, q) \).

The next lemma states that for a run of a dPGD, if there are two executable statements, the execution of one node does not affect the condition \( \text{condC} \) of the other.

**Lemma 5.13.** Let \( s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \ldots \) be a run of a dPGD. For any \( p \) and \( q(p \neq q) \), we assume \( \{p, q\} \subseteq \text{Next}(s_i) \). If \( s_i \xrightarrow{q} s_{i+1} \), the following holds:

\[ \forall r \in G^*(p). r \neq p \Rightarrow \forall (w, r) \in C. ec_{i+1}(w, r) \neq \text{act}. \]  

(5)

Proof. Suppose that claim (5) is violated by the execution of \( q \). By the definition of \( udec \), there exists some \( r \in G^*(p) \) such that \((q, r) \in C \). Since \( \{p, q\} \subseteq \text{Next}(s_i) \), by Lemma 5.8, we have \( p \notin G^*(q) \) and \( q \notin G^*(p) \). By Lemma 5.8, there exists some \( u \in \text{mca}(p, q) \) and \( Q \in \{T, F\} \) such that \( \{p, q\} \subseteq G^*_Q(u) \). Meanwhile, since \( r \in G^*(p) \) and \((q, r) \in C \), there exists some \( o \in G^*(p) - G^*(q) \) such that \( r \in G^*(o) \). That is to say, we have \( r \in G^*(o) \) and \( r \in G^*(q) \). This implies \( \forall v \in \text{mca}(o, q). \forall Q \in \{T, F\}. \{o, q\} \not\subseteq G^*_Q(v) \) from the first condition of the dPGD. However, by Lemma 5.12, we have \( u \in \text{mca}(p, q) \subseteq \text{mca}(o, q) \) which leads to contradiction.

**Lemma 5.14.** Let \( s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \ldots \) be a run of a dPGD and \( q \in \text{Next}(s_i) \). For \( o \in G(q) \) such that \((o, p) \in F \oplus D \), we assume \( ec_i(o, p) = \text{chk} \). Then \( q \) has an outgoing looping edge.

Proof. We assume that all \( C \)-edges from \( q \) are not looping edges. Since \( q \in \text{Next}(s_i) \), there is \((r, q) \in C \) such that \( ec_i(r, q) = \text{act} \). Therefore, there is some \( j < i \) such that \( n_j = r \) and \( \forall k \in (j, i). ec_k(r, q) = \text{act} \). Suppose \( q \in G_T(r) \) (the
case of $G_T(r)$ is the same.) D Since C-edges from $q$ are not looping edges, we have $G(q) \subseteq G_T(r)$. Therefore, $\forall o \in G(q).\forall (o,p) \in F \oplus D. ec_j+1(o,p) = unc$ by the definition of $udec$. Since $r \in \text{Next}(s_j)$, there are no C-edges pointing to a node in $G(r)$ whose states are $act$ at state $s_j$. Moreover, since $G(q) \subseteq G_T(r)$, there are no C-edges pointing to a node in $G(q)$ whose states are $act$ at state $s_{j+1}$. Thus, $q \in \text{Next}(s_{j+1})$. If $s_{j+1} \xrightarrow{T} s_{j+2}$ (i.e., $\{q,v\} \subseteq \text{Next}(s_{j+1})$), by applying Lemma 5.13 we have $\forall o \in G^*(q).o \neq q \Rightarrow \forall (w,o) \in C.ec_{j+2}(w,o) \neq act$. By repeating this argument, we have $\forall o \in G^*(q).\forall (w,o) \in C.ec_k(w,o) \neq act$ for any $k \in (j,i]$. That is to say, no $o \in G(p)$ is executed between $s_j$ and $s_i$. Therefore, $\forall o \in G(q).\forall (o,p) \in F \oplus D. ec_i(o,p) = ec_{j+1}(o,p) = unc$, which leads to contradiction.

The next lemma states that if there are multiple executable statements in a run of a dPDG, execution of one statement does not interfere with executability of the other statements.

**Lemma 5.15.** Let $s$ be a state in a run of a dPDG. For all $p,q \in \text{Next}(s)$ such that $p \neq q$, if $s \xrightarrow{q} s'$, then $p \in \text{Next}(s')$ holds.

**Proof.** We show $\text{condCFLD}(s',p)$ holds. We divide the proof by the type of $q$.

1. If $q$ is an assignment statement, the change of the $ec$ caused by the execution of $q$ does not affect the value of the predicates $\text{condC}$, $\text{condF}$ and $\text{condD}$. Moreover, by Lemma 5.10, $\{p,q\} \subseteq \text{Next}(s)$ implies that there are no $L$-edges between $p$ and $q$. Therefore we have $\text{condCFLD}(s',p)$ and $p \in \text{Next}(s')$.

2. Suppose $q$ is an if statement (ife). We prove each condition individually.

**Proof of condC(s',p).** Since $\text{condC}(s,p)$ holds, there is $(o,p) \in C$ that satisfies $ec(o,p) = act$. If the execution of $q$ brings $ec'(o,p) \neq act$, then $o \in G(q)$ by the definition of $udec$. This, however, contradicts to $\text{condC}(s,q)$ since it implies $p \in G^*(q)$. Therefore, the first conjunct of $\text{condC}(s',p)$ holds. The second conjunct follows from Lemma 5.13.

**Proof of condF(s',p).** Assume that the execution of $q$ causes violation of $\text{condF}(s',p)$. We consider the case $E(q,e)av = T$ (The case of $F$ is the same). By the definition of $udec$, there must be some $r \in G_T(q) - \{q\}$ such that $(r,p) \in F$ and $ec(r,p) = chk_2$. Thus, by Lemma 5.14, $q$ has outgoing looping edges. That is, $q$ is contained in some loop in the CDG by the definition of looping edges. By the third condition of the dPDG, then, $p \in G^*(q)$ is the case, which contradicts to $\text{condC}(s,q)$.

**Proof of condD(s',p).** The same as the proof of $\text{condF}(s',p)$.

**Proof of condL(s',p).** Assume that the execution of $q$ causes violation of $\text{condL}(s',p)$, that is, for some $l(p,r) \in L$, $ec'(l(p,r)) = unc$. We consider the case $E(q,e)av = T$ (The case of $F$ is the same). By the

---

2Note that $q$’s execution brings $ec'(r,p) = unc$, thus $p$ becomes unexecutable at state $s'$. 

26
Lemma 5.16. Let \( s \) be a state of a run of a dPDG. If \( \{p, q\} \subseteq \text{Next}(s) \), then \( G(p) \cap G(q) = \emptyset \) holds.

**Proof.** Assume \( G(p) \cap G(q) \neq \emptyset \). Since \( \{p, q\} \subseteq \text{Next}(s) \), we have \( p \notin G^*(q) \) and \( q \notin G^*(p) \) by Lemma 5.9. Thus, for some \( o_1 \in G(p) - G(q) \) and \( o_2 \in G(q) - G(p) \), there is \( o \in G(p) \cap G(q) \) such that \( (o_1, o) \in C \) and \( (o_2, o) \in C \). Since \( \{p, q\} \subseteq \text{Next}(s) \), there is \( r \in mca(p, q) \) and \( Q \in \{\text{T}, \text{F}\} \) such that \( \{p, q\} \subseteq G^*_Q(r) \) by Lemma 5.8. Since \( o_1 \in G(p) - G(q) \) and \( o_2 \in G(q) - G(p) \), we have \( o_1 \notin G^*(q) \) and \( o_2 \notin G^*(p) \). By applying Lemma 5.12 twice, we have \( r \in mca(o_1, o_2) \) which implies \( \{o_1, o_2\} \subseteq G^*_Q(r) \). This contradicts the first condition of the dPDG. \( \square \)

The next lemma states that if there are two executable statements, the final result of the execution of them are independent of the execution order.

Lemma 5.17. Let \( s \) be a state of a run of a dPDG. If \( \{p, q\} \subseteq \text{Next}(s) \), the following holds:

\[
s \xrightarrow{p} s_1 \xrightarrow{q} s_2 \iff s \xrightarrow{q} s_1' \xrightarrow{p} s_2'.
\]

**Proof.** Suppose \( s \xrightarrow{p} s_1 \xrightarrow{q} s_2 \) and \( s \xrightarrow{q} s_1' \xrightarrow{p} s_2' \). We split the proof by the type of \( p \) and \( q \).

1. Both \( p \) and \( q \) are respectively assignment statements \( x := e_1 \) and \( y := e_2 \).

Since the execution order of \( p \) and \( q \) do not change the final result of the econf, we have \( ec_2 = ec_2' \). If \( x \neq y \), we can easily see \( av_2 = av_2' \) by the definition of udav. We consider the case \( x = y \). If there exists \( u \) such that \( f_x(p, u) \in F \) and \( f_x(q, u) \in F \), it should be \( \{p, q\} \not\subseteq \text{Next}(s) \) by Lemma 5.11 and leads to contradiction. Therefore, \( \{n \mid (p, n) \in F \oplus L\} \cap \{n \mid (q, n) \in F \oplus L\} = \emptyset \). By Lemma 5.10, we have \( (p, q) \notin F \oplus L \) and \( (q, p) \notin F \oplus L \), which yields \( av_2 = av_2' \).

2. \( p \) is an assignment statement \( x := e_1 \) and \( q \) is an if statement \( \text{if}e_2 \). By Lemma 5.9, we have \( p \notin G^*(q) \), and thus \( ec_2 = ec_2' \). Furthermore, we have the following:

\[
av_1 = av[(n, x) \mapsto \mathcal{E}(p, e_1)av : (p, n) \in F \oplus L] = av_2.
\]

\[
av_1' = av_1[(n, x) \mapsto \mathcal{E}(p, e_1)av : (p, n) \in F \oplus L].
\]

Since \( av_1' = av \), and \( (p, q) \notin F \oplus L \) holds by Lemma 5.10, we conclude \( av_2 = av_2' \).
3. Both $p$ and $q$ are respectively if statements $ife_1$ and $ife_2$. By the definition of $udav$, we obviously have $av_2 = av'_2$. By Lemma 5.9, we have $p \notin G^*(q)$ and $q \notin G^*(p)$. By Lemma 5.16, we have $G(p) \cap G(q) = \emptyset$. Therefore, we conclude $ec_2 = ec'_2$.

By the above lemmata, we finally prove the following theorem, which states that executions of a dPDG reach the same final state regardless of execution orders of statements.

**Theorem 5.2.** Let $s$ be a state of a run of a dPDG. If there is a finite run $s \xrightarrow{n_0} s_1 \xrightarrow{n_1} \cdots \xrightarrow{n_m} s_m$, all runs from state $s$ have the length $m + 1$ and end with $s_m$.

**Proof.** By induction on $m$.

**(Base)** The claim is trivial for $m = 0$.

**(Induction)** Suppose that there is a run $s \xrightarrow{p_0} s_1 \xrightarrow{\cdots} s_m$ of the length $m + 1$ starting with $s$. Suppose $Next(s) = \{p, p_0, \ldots, p_j\}$. By Lemma 5.15, we have $\{p_0, \ldots, p_j\} \subseteq Next(s_1)$. Since we have a run $s_1 \xrightarrow{\cdots} s_m$ starting from $s_1$ whose length is $m$, all runs from state $s_1$ have the length $m$ and end with $s_m$ by the induction hypothesis. Therefore, for all $i \in [0, j]$, there is a finite run $s_1 \xrightarrow{p_i} s_1^i \xrightarrow{\cdots} = s_m$ of the length $m$. Since we have $s_0 \xrightarrow{p_0} s_1 \xrightarrow{p_i} s_1^i$ by Lemma 5.17, there is $s_1^i$ such that $s_0 \xrightarrow{p_i} s_1^i \xrightarrow{p_j} s_2$. This implies there is a run starting with $s_1^i$, ending with $s_m$ and whose length is $m$. By the induction hypothesis, all runs from state $s_1$ have the length $m$ and end with $s_m$. Since $i$ is arbitrary, all runs from $s$ have the length $m + 1$ and end with $s_m$. 

5.3 **Equivalence of the operational semantics of CFG and PDG**

Finally, we arrive at the equivalence of the operational semantics of the CFG and the PDG.

**Theorem 5.3.** Let $P$ be a CFG and $G$ be the corresponding PDG. Suppose $\sigma_0 \approx av_0$ and $G$ is a dPDG. If there is a finite run $\sigma_0 \xrightarrow{n_0} \sigma_1 \xrightarrow{n_1} \cdots \xrightarrow{n_m} \sigma_m$ of $P$ where $n_m = (\text{ret } x)$, there is a finite run $s_0 \xrightarrow{n_0} s_1 \xrightarrow{n_1} \cdots \xrightarrow{n_m} s_m$ of $G$ and $\sigma_m(x) = av_m(n_m, x)$.

**Proof.** By Theorem 5.1 and 5.2.

6 **Conclusion**

In this paper, we proposed an operational semantics of the PDG that applies even to unstructured programs. In our operational semantics, a PDG has the same execution sequence as the corresponding CFG (Theorem 5.1). We identified the class of deterministic PDGs (dPDGs) and proved that every run of a
dPDG starting with the same initial state end with the same final state (Theorem 5.2). These facts lead to the semantical equivalence between the CFG and the PDG (Theorem 5.3).

We expect that our operational semantics will be used to formally discuss correctness of program optimisation techniques based on PDG transformations. Since optimised PDGs should be translated into CFGs, there are some algorithms that translate PDGs into CFGs [14, 15, 17]. However, there are cases where optimised PDGs do not have corresponding CFGs. Thus, those algorithms duplicate some nodes or insert artificial conditional statements. Nevertheless, correctness of such transformations are not formally proved. One possible reason is that such PDGs are usually not well-structured and the existing PDG semantics cannot be applied. Since our semantics can be applied to such programs, it will be helpful to prove the correctness of such algorithms.

Another interesting research direction is to prove semantical equivalence between concurrent programs and PDGs. This enables to formally discuss correctness of algorithms that construct concurrent programs from PDGs.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[2] W. Baxter and III H. R. Bauer. The program dependence graph and vectorization. In POPL ’89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 1–11, New York, NY, USA, 1989. ACM Press.

[3] E. Cartwright and M. Felleisen. The semantics of program dependence. In PLDI ’89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation, pages 13–27, New York, NY, USA, 1989. ACM Press.

[4] Jeanne Ferrante and Karl J. Ottenstein. A program form based on data dependency in predicate regions. In POPL ’83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 217–236, 1983.

[5] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[6] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Trans. Program. Lang. Syst.*, 11(3):345–387, 1989.

[7] Sohei Ito, Shigeki Hagihara, and Naoki Yonezaki. An operational semantics of program dependence graphs (in Japanese). *Computer Software*, 26(3):109–135, 2009.
[8] Souhei Ito, Shigeki Hagihara, and Naoki Yonezaki. An operational semantics of program dependence graphs for unstructured programs. In Mitsu Okada and Ichiro Satoh, editors, Advancements in Computer Science - ASIAN 2006. Secure Software and Related Issues, pages 264–271, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[9] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In POPL ’81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 207–218, New York, NY, USA, 1981. ACM.

[10] Ikuo Nakata. Compiler Construction and Optimization (in Japanese). Asakura Publishing, 1999.

[11] R. Parsons-Selke. A rewriting semantics for program dependence graphs. In POPL ’89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 12–24, New York, NY, USA, 1989. ACM Press.

[12] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. IEEE Trans. Softw. Eng., 16(9):965–979, 1990.

[13] G. Ramalingam and Thomas Reps. Semantics of program representation graphs. Technical Report CS-TR-1989-900, University of Wisconsin, Madison, 1989.

[14] B. Simons, D. Alpern, and J. Ferrante. A foundation for sequentializing parallel code. In SPAA ’90: Proceedings of the second annual ACM symposium on Parallel algorithms and architectures, pages 350–359, New York, NY, USA, 1990. ACM Press.

[15] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft, Redmond, WA, 1993.

[16] Joe Warren. A hierarchical basis for reordering transformations. In POPL ’84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 272–282, New York, NY, USA, 1984. ACM Press.

[17] Jia Zeng, Cristian Soviani, and Stephen A. Edwards. Generating fast code from concurrent program dependence graphs. In LCTES ’04: Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, pages 175–181, New York, NY, USA, 2004. ACM Press.