Causal Program Dependence Analysis

Seongmin Lee
KAIST
Daejeon, Republic of Korea
bohrok@kaist.ac.kr

Dave Binkley
Loyola University Baltimore
Baltimore, USA
binkley@cs.loyola.edu

Robert Feldt
Chalmers University of Technology
Göteborg, Sweden
robert.feldt@chalmers.se

Nicolas Gold
University College London
London, UK
n.gold@ucl.ac.uk

Shin Yoo
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

ABSTRACT
We introduce Causal Program Dependence Analysis (CPDA), a dynamic dependence analysis that applies causal inference to model the strength of program dependence relations in a continuous space. CPDA observes the association between program elements by constructing and executing modified versions of a program. One advantage of CPDA is that this construction requires only light-weight parsing rather than sophisticated static analysis. The result is a collection of observations based on how often a change in the value produced by a mutated program element affects the behavior of other elements. From this set of observations, CPDA discovers a causal structure capturing the causal dependence between program elements. Qualitative evaluation finds that CPDA concisely expresses key dependence relationships between program elements. As an example application, we apply CPDA to the problem of fault localization. Using minimal test suites, our approach can rank twice as many faults compared to SBFL.

1 INTRODUCTION
Program dependence analysis is a fundamental task in software engineering. In addition to being the foundation of program comprehension [45], program dependence analysis underlies various software engineering tasks, including software testing [8], debugging [21, 25], refactoring [12], maintenance [15], and security [22]. It often supports these tasks by reducing the number of program elements that must be considered.

Each component of a program depends, to some degree, on the other components of the program. Some of these dependence relations are strong, i.e. a change in one component almost surely changes the value of another, while others are weaker. Traditional static dependence analysis attempts to safely approximate all possible dependence relations. This can become quite involved, and will return many relations. A good example would be pointer analysis, which is not only computationally expensive but also prone to produce a large number of dependence relations including many false positives. Furthermore, static analysis typically does not attempt to estimate the strength of a dependence. Thus its output is a large number of equally-weighted dependencies, which tends to degrade the ability of a developer to work with the code based on the results of the analysis. A few dynamic dependence analysis approaches have modeled dependence strengths [6, 44], but they are typically built on top of (expensive) static dependence analysis (e.g., the construction of a Program Dependence Graph), and remain vulnerable to confounding bias due to their formulation based on conditional probability, which captures mere association, rather than causation [6, 26, 44].

This paper proposes Causal Program Dependence Analysis (CPDA), a framework that analyzes the degree (or strength) of the dependence between program elements. Given a set of executions, CPDA makes use of associations between observed behavior of program elements to discover a causal structure. By applying techniques for causal inference [31, 33], over this causal structure, CPDA then produces two measures of program dependence: causal dependence and direct dependence. Causal dependence aggregates the effect that a change at a program element has on the behavior of other program elements. A direct dependence is the effect of one program element on another, excluding any effects that pass through other elements.

One advantage of CPDA is that it does not require extensive “heavy weight” static analysis. Instead, only light-weight parsing of the source code, or light-weight instrumentation of a binary, is required. Essential points of interest in the code are annotated so we can introduce simple mutations to the state, and observe their effects using a set of test inputs. An additional benefit is thus that our approach can be applied to heterogeneous systems built from multiple languages, as long as a parser (or method for instrumenting binary representations) is available.

To evaluate CPDA, we first build a Causal Program Dependence Model (CPDM), and then introduce and study its use for Causal Dependence based Fault Localization (CDFL). The CPDM is a graph in which we annotate the direct dependence on the causal structure to represent a program’s dependences. We investigate how well the (strength of the) direct dependence captures the program’s semantics. We also consider how characteristics of the test suite impacts the resulting CPDM.

To further study the utility of CPDA, we propose Causal Dependence based Fault Localization (CDFL), which calculates the suspiciousness of program elements in light of failing test cases, based on CPDA. We evaluate CDFL using the Siemens suite, a well-known benchmark for testing and debugging. Compared to Spectrum based Fault Localization (SBFL) [29, 43], our empirical results CDFL can rank 166% more actually faulty program elements at the top.

The main contributions of this paper include the following:

- We propose Causal Program Dependence Analysis. This dependence analysis framework, based on causal inference, supports the quantification of dependence strength among program elements.
Causal Program Dependence Analysis (CPDA) aims to model and quantify the strength of program dependence relations. The dependence reported by CPDA is thus not binary: rather, it represents how likely a change to the value of a program element $S_i$ is to change the value of another element $S_j$. To capture the causal relations between program elements, we initially capture the behavior of the original program when executed on a test suite, to use as oracle. We then observe whether systematically mutated program variants behave differently to the oracle. Technically, we let the change status of a program element be a random variable in causal inference. Our causal inference thus uses both purely observational data, the behavior of the unmutated program, as well as interventional data, i.e. when we have taken actions that lead to changed values for some subset of factors of a DAG. A hallmark of causal analysis is that its DAGs can be used both to guide which factors to intervene on, i.e. change, and then tell us how to calculate the causal effects from our observations.

2.2 Overview of CPDA

Causal Program Dependence Analysis (CPDA) identifies target program elements to analyze, and subsequently mutates and instruments the code (Sec. 2.3). The instrumentation allows us to generate an oracle, a set of recorded behaviors, i.e. states of variables, of the original, unmutated program. Supermutation allows us to apply interventions to the program elements and monitor the resulting changes [41]. Using these observations (Sec. 2.4), CPDA builds a causal structure (Sec. 2.5) and performs causal inference to calculate the causal and the direct dependence (Sec. 2.6). We also introduce Causal Dependence based Fault Localization (CDFL), a novel fault localization technique based on CPDA. Using causal dependence, we try to identify the program element that is the most likely to cause the observed failure (Sec. 2.7).

2.3 Instrumentation, Oracle, and Mutation

We mutate and instrument the target program simultaneously, so that we can apply intervention to the behavior of the original
program and observe its impact. Given a program, let a node correspond to each left-hand side variable that occurs in assignment statements, function parameters, predicates, and return values of functions: we use the term “node” because CDPA eventually will represent all these program elements as nodes in the graph of a causal model. Given a node, its trajectory represents the behavior of the node, i.e. the sequence of values it takes during execution.

2.3.1 Instrumentation. A pair consisting of an intervention and the measurement of its impact is called an observation. To efficiently produce a large and diverse set of observations for program \( P \), we construct a super mutant [41], i.e., a meta-mutated program that takes a mutation position (a unique index of each node), and a mutation value that will be used whenever the node is executed, as input: this reduces the number of compilations required for multiple mutations. The instrumentation begins by indexing all the nodes in the target program using a parser, and injecting a helper function. The helper function, when given a target node index, will overwrite (i.e., mutate) the value of the node, and log the result of mutation, during execution. For all the other nodes, the helper function simply logs their current value observed during execution (i.e., record their trajectories).

2.3.2 Oracle. Once the target program is instrumented, we can obtain the oracle for each node using the given test suite. An oracle for a node is simply a collection of all of its trajectories, one per a test input in the test suite, when no mutations are active during execution.

2.3.3 Mutation. CPDA currently targets primitive variables of the following types: bool, char, int, long, float, double, and string. For booleans, we mutate by simply negating the original value. For primitive types, we sample a random value from a Gaussian distribution that is based on all observed values in its trajectories. However, when the domain knowledge clearly specifies the range of possible values (such as an input parameter that can be only 0, 1, or 2), we simply sample from the given range with a uniform probability. Finally, for strings, we first sample the string length from a Gaussian distribution that is based on the length of all observed strings for the node, and subsequently sample a random lowercase string of that length. We sample the value mutation up to \( N_{mpn} \) times per node.\(^2\) We leave more refined data mutation and generation strategies for future work and note that techniques for test generation can likely be of use as well as able to handle more complex, structured data types [13].

2.4 Representing Behavioral Changes

Once we collect the oracle (trajectories obtained by executing the original program with the given test suite) and our observations (trajectories obtained by executing the mutated programs with the given test suite), we can identify the change of program behavior due to value mutations. Formally, we define change of the behavior for a node as follows:

**Definition 2.1 (Change of the behavior of a node).** Behavior of node \( S_i \) in program \( P \) has changed for a given mutation and an input, if the trajectory for \( S_i \) using the mutated program \( P' \) is different from the trajectory of \( S_i \) using \( P \).

From now on, let us overload the notation \( S_j \) to represent both the node itself and the variable denoting whether the behavior of \( S_j \) has changed under \( P' \) or not; \( S_j = 1 \) if changed, 0 otherwise. For a fixed mutated version \( P' \) and an input, we get a single observation, which is a boolean vector whose length is equal to the number of nodes in the program: each value in the observation indicates whether the behavior of the corresponding node has changed or not. Given a set of observations, we can calculate the probability of a node changing its behavior.

**Definition 2.2 (Probability of the behavior change).** Given Node \( S_i \) from Program \( P \) and a set of observations \( O \), \( P_O(S_i = 1) \) represents the probability of a change in \( S_i \)’s behavior, given \( O \), is:

\[
P_O(S_i = 1) = 1 - P_O(S_i = 0) = \frac{|\{obs \in O \mid obs[S_j] = 1\}|}{|O|}.
\]

Moreover, we can also express the conditional probability of node \( S_j \) changing its behavior given that node \( S_j \) has changed its behavior:

**Definition 2.3 (Conditional probability of the behavior change).** For a given set of observations, \( O \), behavior, and \( O' = \{o \mid o \in O \land S_j \neq \text{mutated in the program that generates } o\} \):

\[
P_O(S_j = 1 \mid S_i = 1) = \frac{P_O(S_j = 1 \land S_i = 1)}{P_O(S_i = 1)} = \frac{|\{obs \in O' \mid obs[S_j] = 1 \land obs[S_i] = 1\}|}{|\{obs \in O' \mid obs[S_i] = 1\}|}.
\]

The conditional probability in Definition 2.3 becomes the foundation of causal dependencies that we calculate later. We exclude the mutation on \( S_j \) itself, hence \( O' \), when calculating the conditional probability \( P_O(S_j = 1 \mid S_i = 1) \). This is to avoid negating the effect from the change of \( S_i \) with a mutation of \( S_j \) itself.

Each node can have up to \( N_{mpn} \) mutated values (see Section 2.3). Since each node may have a different number of mutated values, we normalize the probabilities in Def. 2.2 and 2.3 by multiplying the reciprocal of the number of sampled mutated values. For example, each observation for an integer type node mutation from ten possible values (such as an input parameter that can be only 0, 1, or 2), we simply sample from the given range with a uniform probability. For a node changing its behavior.

2.5 Causal Structure Discovery

The conditional probability defined in Def. 2.3 expresses the association between changes of program elements: changes are simply observed together. To elevate this to causal inference, we need the concept of a specific change preceding another. It is also necessary to distinguish between direct predecessors, i.e., nodes whose change directly affects our target node, and non-direct predecessors, i.e., nodes whose change does affect our target node but indirectly through one or more direct predecessors.

A causal structure allows us to introduce this concept of precedence. Program dependence is inherently a form of causal precedence: if node \( S_j \) depends on node \( S_i \), a change in \( S_i \) will precede the change in \( S_j \). Consequently, in principle, a perfectly accurate PDG of the target program can serve as a causal structure, if available. However, in reality, existing dependence analysis is both computationally costly and may not be accurate. Instead, we use ideas from

---

\(1\) Some ambiguous types in the subject programs have been resolved manually

\(2\) Due to boolean nodes or nodes with predetermined value ranges, we may not be able to sample the same number of mutated values for all nodes.
the causal inference field and approximate the causal structure from the set of observations we have collected.

Let us first formally define the predecessors of a node. We identify the predecessors of a node $S_j$ by checking whether intervening on a particular node affects $S_j$. We call predecessors of a node as its \emph{intervention parents}.

\textbf{Definition 2.4 (Intervention Parent).} For a program $P$ and a set of nodes $S$ from the program, a set of nodes $IPA_j \subseteq S$ is said to be intervention parents of $S_j$ if mutating any node in $IPA_j$ changes the behavior of $S_j$ for at least one input. In other words, if a mutated program $P'$ that changes the value of $S_k$ to $v$ causes $S_j = 1$ under input $i$, then $S_k \in IPA_j$.

However, the precedence relationship may be direct, or indirect (i.e., a change in one of the intervention parents will, by definition, precede the change in the child node, but its effect may or may not need to pass through another parent to affect the child node). Based on the theory of causal inference, we define Markovian parents of a node as the minimal set of direct predecessors among the intervention parents.

\textbf{Definition 2.5 (Markovian parent).} The Markovian parents of $S_j$, $PA_j$, is a minimal set of predecessors of $S_j$ that renders $S_j$ independent of all its intervention parents. In other words, $PA_j$ is any subset of $IPA_j$ such that it satisfies $P(s_j \mid pa_j) = P(s_j \mid ipa_j)$ while no other proper subset of $PA_j$ does.\footnote{Lowercase symbols (e.g., $x$, $pa_j$, and $ipa_j$) denote particular realizations of the corresponding variables (e.g., $X$, $PA_j$, and $IPA_j$).}

Subsequently, the Markovian parent-child relation constructs the causal structure of the program.

\textbf{Definition 2.6 (Causal structure).} The causal structure of a program represents the Markovian relation between the nodes. The causal structure is represented by a graph, $G = (V, E)$, where $V = \{S_i\}_{i \in 1..n}$ and $E = \{(S_i, S_j) \mid S_i \in PA_j\}$, where $PA_j$ denotes Markovian parents of $S_j$.

It requires an exponential number of computations to compute the conditional probability in Def. 2.5 for every possible combination of intervention parents. Thus, we approximate the Markovian parents by iteratively removing non-Markovian parents from intervention parents. Algorithm 1 shows the process of removing non-Markovian parents from $IPA_j$. We choose one node $S_d$ from $IPA_j$ and check whether $S_d$ is independent from $S_j$, given all other candidate nodes.

If $S_d$ is the only candidate node left (Line 7-9), we check whether $S_j$ is independent from $S_d$. If it is, $S_d$ is not a Markovian parent, otherwise, $S_d$ is not a Markovian parent. If there are other candidate nodes $s_{other}$ (Line 11-16), we check the conditional independence of $S_j$ from $S_d$ for every realization of $s_{other}$. If $S_j$ is conditionally independent from $S_d$ for all $s_{other}$, $S_d$ is not a Markovian parent. Otherwise, $S_d$ could be a Markovian parent of $S_j$. This is because $S_j$ can be conditionally independent from $S_d$ if $s_{other}$ changes. Therefore, we re-check all possible Markovian parents every time we find a new non-Markovian parent. To minimize re-checking cost, we order the candidate nodes ($Dist$) to choose the most unlikely to be the Markovian parent first (Line 5). Our intuition is that, during the execution of the program, the node executed earlier than the execution of $S_j$ is less likely to be a parent of $S_j$. Algorithm 1 approximates Markovian parents of a node, as it only considers individual nodes, and not their combination, for Makovian parents.

The last step of building a causal structure is to eliminate cycles in the structure. In causal inference, the form of the causal structure is the \emph{Bayesian network}, which has no cycles. To eliminate cycles, we remove from $S_j$ in $PA_j$ if $S_j$ is in $IPA_i$.\footnote{We further discuss the cyclic relation in the causal model in Section 5.}

\section{6 Causal Program Dependence Model}

We first introduce causal dependence, which measures the total effect of each node’s change that causes a change in another node, and subsequently introduce direct dependence, which measures the effect of one node on another excluding all the (indirect) effects through other nodes.

\subsection{2.6 Causal dependence.} The conditional probability $P(\cdot \mid x)^3$ represents the probability when one observes that $X$ has value $x$. By having a causal structure, we can instead estimate the probability when we \emph{force} $X$ to have the value $x$, denoted as $P(\cdot \mid do(x))$. The difference between “forcing” and “observing” comes from whether and how the effect of $X$ is affected by other nodes. A \emph{confounding bias}, a distortion representing the event that is associated but not casually related to the observation, appears through the so-called \emph{backdoor path} in a causal DAG [31]. By ignoring the incoming effect
of $X$, we can remove the effect through the backdoor path to $X$, subsequently eliminating the confounding bias from the association between $X$ and another node $Y$. Note that backdoor paths in CPDM are based on the inferred causal structure, unlike existing work [6]. Based on Pearl et al. [31], the causal effect estimating $P(\cdot | do(x))$ is formally defined below: we use a set of nodes, instead of a single node, to include the more general scenario where multiple nodes are mutated simultaneously.

**Definition 2.7 (Causal Effect).** Let $G = (V, E)$ be a causal structure. Given two disjoint sets of nodes, $X, Y \subseteq V$, the causal effect of $X$ on $Y$, denoted either as $P(y \mid do \circ X)$ or $P(y \mid do(X))$, is a function from $X$ to the space of probability distributions on $Y$. For each realization $x$ of $X$, $P(y \mid \hat{x})$ gives the probability that $Y = y$ induced by deletion from the causal structure of all the edges to nodes in $X$ other than $x$ and thus considering $X = x$. The causal effect $P(y \mid do(X))$ is calculated as:

$$P(y \mid do(x)) = \sum_{p_{X}} P(y \mid x, p_{X})P(p_{X})$$

where $p_{X}$ represents the Markovian parents of $X$.

In causal program dependency analysis, we aim to measure the amount of impact that the change of one node $S_i$ has on another node $S_j$, i.e., the difference between the causal effect from $S_i = 1$ to $S_j$, and from $S_i = 0$ to $S_j$. Thus, we introduce causal dependence as follows:

**Definition 2.8 (Causal Dependence).** The causal dependence of node $S_i$ to $S_j$ measures how much changing the behavior of $S_i$ affects the behavior of node $S_j$. Given a set of observations $O$, the causal dependence of $S_i$ to $S_j$, $CD_{O}(S_i, S_j)$, is defined as:

$$CD_{O}(S_i, S_j) = P_{O}(S_j = 1 \mid do(S_i = 1)) - P_{O}(S_j = 1 \mid do(S_i = 0)).$$

### 2.6.2 Direct dependence

**Direct Dependence.** Direct dependence quantifies the amount of effect that is not mediated by, any nodes. More formally, it measures the sensitivity of $Y$ to changes in $X \in \{\text{Markovian parents of } Y\}$ while all other Markovian parents of $Y$ are held fixed. Based on Pearl et al. [31], a generic definition that fits such sensitivity is the natural direct effect.

**Definition 2.9 (Natural Direct Effect).** The natural direct effect, denoted either as $NDE_{X \rightarrow \hat{x} \rightarrow Y}(Y)$ is the expected change in $Y$ induced by changing $X$ from $x$ to $x'$ while keeping all mediating factors constant at whatever value they would have been under $do(X)$. The natural direct effect, $NDE_{X \rightarrow \hat{x} \rightarrow Y}(Y)$, is calculated as:

$$\sum_{Z} E(Y \mid do(x', z)) - E(Y \mid do(x, z)) | P(z \mid do(x)),$$

where $Z$ represents all parents of $Y$ excluding $X$.

Similar to causal dependence, in the context of CPDA, we estimate how much effect a parent has on a child node $S_j$ when the parent node $S_i$ alter from “unchanged” to “changed”, regardless of all other parents of $S_j$. Based on the definition of the natural direct effect, we define the direct dependence as follows.

**Definition 2.10 (Direct Dependence).** The direct dependence of $S_j$ from $S_i$, denoted as $DD_{F}(S_i, S_j)$, is the average of the natural direct effect of $S_j$ towards $S_j$ over all inputs $I$:

$$DD_{F}(S_i, S_j) = \frac{1}{|I|} \sum_{t \in I} NDE_{O_t, S_i, 0\rightarrow 1}(S_j),$$

where $O_t$ denotes the observations from input $t$, and $NDE_{O_t, S_i, 0\rightarrow 1}(S_j)$ denotes the natural direct effect using the observations $O_t$.
Table 2: Example comparing CDFL to Dynamic Slicing and Dicing. ‘␣’ denotes the whitespace.

| Code | Coverage | DS | Dice | SusPct. |
|------|----------|----|------|---------|
| s = input() | 1 1 1 1 | 0 | 1.0 - 0.0 = 0.0 | |
| pred = isEndSpace(s) | 0 1 1 1 | 0 | 1.0 - 0.5 = 0.5 | |
| if (pred) p = nさてぬ p.strip() | 0 1 1 1 | 0 | 1.0 - 0.5 = 0.5 | |
| return p | - - - - | - | - | - |

Test Results P F P F

2.7.2 Advantages over Dynamic Slicing and Dicing. Table 2 contains a motivating example showing the advantages of CDFL over DS, a dynamic backward slice of the returned variable p, and a dice [3], which essentially returns the set difference between dynamic slices computed using failing test inputs and passing test inputs. Intuitively, if there is a dependence relationship that is only exercised in failing executions, dicing reports it as the likely root cause of the failure. The original code in Table 2 intends to only remove trailing whitespaces (denoted as ‘␣’), but the faulty version strips all whitespaces. Because one of the passing inputs, “␣”, and the failing input, “␣”, execute all lines of the code, dynamic slicing reports all lines to be equally faulty, while dicing concludes that none of them are faulty. However, CDFL reports that the faulty line is more suspicious, based on the quantitative dependence.

3 EXPERIMENTAL SETUP

This section presents our research questions and the set-up of our empirical evaluation.

3.1 Research Questions

CDFL is but one possible use of the general program dependence analysis method of CPDA. In addition to evaluating CDFL, we thus first ask a more general research question about CPDA.

RQ1. Effectiveness of CPDA: How well does CPDA capture the relative strengths of dependence between program elements? We expect the direct dependence to capture the varying degrees of dependence between program elements. To investigate this, we build CPDMs for an example program with well known dependencies, and examine how it differs from PDG. We also investigate its sensitivity when using different test suites.

RQ2. Performance of CDFL: How well does CDFL perform compared to other, widely studied fault localisation techniques? We evaluate the utility of CPDA through CDFL, by comparing the fault localisation results to Spectrum Based Fault Localization (SBFL), dynamic slicing, and dicing. We also investigate the impact that the number of observations, N_{mpn}, has on the accuracy of CDFL.

3.2 Subjects and Baselines

This section describes the subject programs, and baseline techniques, for the CPDA (RQ1) and CDFL (RQ2) study, respectively.

3.2.1 CPDA Study (RQ1). To investigate RQ1, we apply CPDA to the word count (wc) program, which has been widely studied for program dependence analysis [15, 26]. Figure 2 shows the pseudo-code of the word count program. The node index annotations in angled brackets are used to refer to corresponding nodes in Section 4.1. The word count program takes a text as input and counts its number of characters, lines, and words. We use a test suite of 15 test cases, which can be grouped into the following: onechar: Tests 1-3 contain a single letter. oneword: Tests 4-7 contain a single word of multiple letters. oneline: Tests 8-10 contain a single line with multiple words. multoline: Tests 11-15 contain multiple lines of multiple words. For each node, we use a sample of 100 mutations (N_{mpn} = 100) and observe the changes at other nodes.

Figure 2: Pseudo-code of the word count program

| Code | Coverage | DS | Dice | SusPct. |
|------|----------|----|------|---------|
| 1 def main() { | 10 | 13 | inword + 1 | |
| 2 if (characters = 0) | 19 | - | - | |
| 3 <2>lines = 0 | 20 | else | - | - |
| 4 if (words = 0) | 21 | - | - | |
| 5 if (word = 0) | 22 | - | - | |
| 6 if (pred) getChar(<0>)c | 23 | - | - | |
| 7 while (pred) { | 24 | - | - | |
| 8 <7>characters = characters + 1 | 25 | def isLetter(<17>c) { | |
| 9 <8>pred2 = c == 'n' | 26 | isLetter(<17>c) { | |
| 10 if (pred2) | 27 | (c == 'a' && c <= 'z') | |
| 11 <9>lines = lines + 1 | 28 | <10>pred5 = isLetter(c) | |
| 12 <10>pred3 = isLetter(c) | 29 | 10<ret = True | |
| 13 if (pred3) { | 30 | - | - | |
| 14 <11>pred4 = inword == 0 | 31 | <12>ret = False | |
| 15 if (pred4) | 32 | - | - | |
| 16 <12>words = words + 1 | 33 | return _ret | |
| 17 } | - | - | - | |

Table 3: Statistics of the Siemens suite

| Subject | SLoC | N_{bug} | N_{node} | N_{failing test} | N_{node} |
|---------|------|--------|---------|-----------------|---------|
| tces | 135 | 34 | 5-7 | 1-1 | 43 |
| schedule | 292 | 4 | 8-9 | 1-2 | 94 |
| schedule2 | 297 | 3 | 7-10 | 1-1 | 147 |
| totinfo | 346 | 18 | 7-10 | 1-4 | 172 |
| printtokens | 475 | 3 | 11-17 | 1-1 | 293 |
| printtokens2 | 401 | 6 | 12-15 | 1-3 | 186 |
| replace | 512 | 24 | 16-20 | 1-2 | 318 |
| total | 2,458 | 92 | - | - | 1,253 |

3.2.2 CDFL Study (RQ2). We compare CDFL to three existing FL techniques: Spectrum Based Fault Localization (SBFL), Dynamic Slicing (DS), and Dicing. As SBFL technique, we choose Ochiai [30] which is widely studied in the literature [1].

We compute dynamic slices using the dynamic slicing algorithm of Agrawal and Horgan [2], which intersects coverage information from a set of inputs with a static slide. We employ gcov to collect coverage information, and CodeSurfer, a static analysis tool produced by GrammaTech [17], for static backward slices. Finally, we compute the dice as the difference of the dynamic slices of the failing and passing tests. Note that CDFL ranks nodes, while SBFL ranks program statements.

We compare four fault localization techniques using programs from the Siemens suite [11]. Table 3 provides descriptive statistics of the studied programs: columns represent Non-comment-non-blank Lines of Code (SLoC), number of studied bugs, tests, failing tests, and nodes, respectively. We randomly choose a statement coverage adequate test suite provided by SIR that contains at least one failing test for each bug.
Figure 3: Program Dependence Graph (PDG) and Causal Program Dependence Models of word count with different thresholds for the Direct Dependences (DD). For CPDMs, the thicker and darker the edge is, the larger the direct dependence.

Some of the seeded faults provided by SIR have been excluded from our study. Excluded faults can be categorized into four groups. First, we exclude omission faults, because their root causes do not exist in the given source code. Second, we exclude faults for which no failing test case exists. Third, we exclude faults that result in nondeterministic behavior, or segmentation faults, as they produce either inconsistent, or no coverage information. Finally, we exclude any fault whose root cause falls outside the scope of CPDA (for example, if the fault is calling a wrong function without storing the return value, we cannot represent the fault as a node in CPDA).

For evaluation, we use the widely adopted acc@n metric, which counts the number of faults whose root causes are ranked within the top n places. If there are multiple faulty elements, we compute acc@n using the most highly ranked element. Since SBFL techniques tend to produce a lot of ties, we report rankings obtained by multiple tiebreakers. The average tiebreaker computes the average rank of the tied candidates. Min and max tiebreakers rank all tied elements at the highest, and the lowest, place, respectively. Finally, line order tiebreaker breaks ties independently from score distributions using line numbers, facilitating fair comparison between rankings theoretically [43].

3.3 Implementation & Environment

For node selection, we use the open source C static analyzer Frama-C [23]. To insert logging functions in the original program, we use srcML [10], an open source tool that parses and converts source code into the XML format for manipulation.

The Markovian parent calculation (Algorithm 1) can be costly when the number of intervention parents and the number of observations are large. To reduce the cost of calculating the Markovian parents, we use safe parents (SPA), which is a subset of the intervention parents. Given IPA_{j}, the intervention parent set of S_{j}, the safe parent set SPA_{j} is calculated by following heuristics:

- SPA_{j} contains all the nodes in the same method as S_{j} (M_{j}) or the parameters of the methods calling M_{j}.
- If S_{j} is the left-hand side of an assignment statement, and if the assignment statement invokes the function f, SPA_{j} contains the return nodes of f.
- If S_{j} uses a global variable, SPA_{j} contains all the nodes of the global variable.

While using the safe parent set requires an extra call graph analysis, it reduces the Markovian parent calculation time five-fold. Our experiments were performed under CentOS Linux 7, on an Intel(R) Xeon(R) CPU E5-2630 v4 with 250GB of memory.

4 RESULTS

This section answers the research questions based on the results of our empirical investigation of CPDA.

4.1 Effectiveness of CPDM

We first consider the Causal Program Dependence Model created using the entire test suite of 15 test cases and then the four models created using the four groups. Figure 3 contains the PDG of wc produced manually (Fig. 3(a)), along with visualizations of three CPDMs that contain edges with different ranges of direct dependence (DD) values. The thickness of edges corresponds to its direct dependence. Figure 3(b), 3(d), and 3(c), show edges with direct dependence less than 0.2, between 0.2 and 0.8 (non-inclusive), and greater than 0.8, respectively.

4.1.1 Program Comprehension with Quantifiable Dependence. Filtering out edges based on their direct dependences reveals patterns that relate to features in the source code. Consider Figure 3(c) that only shows edges with strong direct dependence. Nodes connected in such edges often correspond to features that are executed in all executions. For example, nodes (5), (15), and (16), that are connected in Figure 3(c), collectively implement the function of reading a character from the input file. Similarly, nodes (1) and (7) increase the character count in each iteration. Nodes (2), (8), and (9) increase the line count, while (10), (13), and (14) determine if the current character is part of a word.

Figure 3(d) shows the edges with intermediate direct dependence values between 0.2 and 0.8. Compared to Figure 3(c), the connected nodes often reflect features that are only occasionally executed. For example, the word counting feature is only executed when there is at least one non-alphabet character in the input. Since seven out of 15 tests (three single character tests and four single word tests) do not have non-alphabet characters, the nodes implementing word counting feature consequently have small direct dependences for the whole test suite. Such dependences (denoted [from] → [to]) include {20} → {10} and {14} → {11}. Even a feature that is executed in every execution may have a small direct dependence.
Figure 4: Differences in the CPDMs for word count created using the four different partitions of the test suite. In each figure A − B, shows edges in A only in solid-red, edges in B only in dashed-blue, and edges in both in gray.

For example, the character read at node ⟨16⟩ affects the predicate at node ⟨8⟩, which in turn checks whether the character is a newline or not. Since among many characters, only newline character makes the difference, their causal relationship is not strong.

We posit that the capability to focus on different bands of direct dependencies can help program comprehension. Despite being a small program of only about 40 lines, its PDG with equally weighted edges makes it difficult to understand the program semantics. To highlight the differences between PDG and CPDM, we discuss a few edges that are not found in the CPDM and why there are good, semantically meaningful reasons for this.

- {⟨5⟩} → {all nodes control dependent on ⟨5⟩ except ⟨15⟩}: ⟨5⟩ and ⟨15⟩ are the predicates of the main loop. Therefore, all nodes within the loop are control dependent on the two. Since ⟨15⟩ is in the loop, if ⟨5⟩ changes, ⟨15⟩ also changes (along with all other nodes in the loop). In this case ⟨5⟩ is considered not as a direct parent of any of the other nodes in the loop and, consequently, CPDM produces the edge ⟨{⟨5⟩} → {⟨15⟩}⟩ while the PDG has many edges going out from ⟨5⟩.

- {⟨15⟩} → {⟨8⟩, ⟨10⟩, ⟨17⟩}: ⟨8⟩, ⟨10⟩, and ⟨17⟩ are control dependent on ⟨15⟩. However, they are also data dependent on ⟨16⟩, ⟨⟨19⟩, ⟨20⟩⟩, or ⟨16⟩, respectively, which in turn depend on ⟨15⟩. If ⟨15⟩ changes, ⟨16⟩, ⟨19⟩, and ⟨20⟩ changes before ⟨8⟩, ⟨10⟩, and ⟨17⟩. Thus, the corresponding PDG edges are not in CPDM.

Let us now consider the relative strengths of dependence represented by the edge weights in CPDM. We highlight the following:

- {⟨19⟩} → {⟨10⟩} > {⟨20⟩} → {⟨10⟩}: In function isLetter, ⟨19⟩ and ⟨20⟩ are the nodes that express whether the character is an alphabetic character or not. Since seven out of the 15 tests do not have a non-alphabet character, the effect on ⟨10⟩, which receives a return value of isLetter, of ⟨19⟩ is more significant than that of ⟨20⟩. It is important to keep in mind that the CPDM takes into account if a node’s behavior changes during execution, but not the number of different values it produces. Thus it is the number of tests cases and not their length that is important here.

- {⟨4⟩} → {⟨11⟩} > {⟨13⟩} → {⟨11⟩} > {⟨14⟩} → {⟨11⟩}: ⟨4⟩, ⟨13⟩, and ⟨14⟩ all correspond to the variable inword. ⟨4⟩ has the highest direct dependence on ⟨11⟩ because it affects whenever there exist more than zero alphabet characters. ⟨13⟩ and ⟨14⟩ affect ⟨11⟩ if the input includes more than one alphabet character. Because there are test cases that do not have any non-alphabet characters, ⟨15⟩ affects ⟨11⟩ more than ⟨14⟩.

### 4.1.2 Impact of Test Suites on CPDM

Let us consider the impact of building the CDPM for wc using the four categories of its test suites: onechar, oneword, oneline, and multiline. Figure 4 shows differences in the CPDMs constructed using different categories. The structure of CPDM changes according to the used category. In a figure labeled A − B, solid red edges are found only in A, dashed blue edges are found only in B, and gray edges are found in both A and B. We highlight some of the findings below.

- One character versus multiple characters (Figure 4(a)): ⟨15⟩, ⟨16⟩, and ⟨13⟩ affect other only when there are more than one characters in the input. Therefore, there are outgoing edges from these in the CPDM of oneword but not in onechar. Instead, the direct predecessor nodes of ⟨15⟩, ⟨16⟩, and ⟨13⟩ ⟨{⟨5⟩}, ⟨6⟩, and ⟨10⟩⟩ are linked to the nodes affected by ⟨15⟩, ⟨16⟩, and ⟨13⟩ in onechar.

- One word versus multiple words (Figure 4(b)): The sequence of edges ⟨{⟨18⟩} → {⟨20⟩} → {⟨10⟩} → {⟨14⟩} → {⟨11⟩}⟩ is introduced in the CPDM created from oneline: the model reflects the information flow from reading a non-alphabet character to the increment of the word counter. Comparison with Figure 4(a) shows that the direct dependence of ⟨{⟨4⟩} → {⟨11⟩}⟩ and ⟨{⟨13⟩} → {⟨11⟩}⟩ have decreased due to the advent of a new node, ⟨14⟩, affecting ⟨11⟩. Note that, unlike in Figure 3(d), the direct dependence of ⟨{⟨19⟩} → {⟨10⟩}⟩ and ⟨{⟨20⟩} → {⟨10⟩}⟩ is similar. This is because every test in oneline contains both alphabet and non-alphabet characters.

- One line versus multiple lines (Figure 4(c)): With a newline character in the input, ⟨{⟨15⟩} → {⟨8⟩}⟩ disappears, while ⟨{⟨16⟩} → {⟨8⟩}, {⟨8⟩} → {⟨9⟩}, and ⟨{⟨2⟩} → {⟨9⟩}⟩ appear: the new edges represent counting of lines. Furthermore, using multiline makes the effect of ⟨16⟩ on ⟨8⟩ larger than that in Figure 3(d), because every test case contains a newline character.

**Answer to RQ1:** The numerical causality associated with each dependence tells us the strength of the causal connection between the two program elements involved. Our investigation shows that by exploiting these weights, CPDM can effectively focus a developer on dependencies of greater importance (those with higher causality). This focus provides a better understanding of the program.
Table 4: Acc@n for CDFL (N_{mpn} = 20, median), SBFL, Dicing, and DS. Avg., LO, Max, and Min are different tiebreakers.

| n  | CDFL Avg | SBFL Avg | DS Avg | Dicing Avg | CDFL LO | SBFL LO | DS LO | Dicing LO | CDFL Max | SBFL Max | DS Max | Dicing Max | CDFL Min | SBFL Min | DS Min | Dicing Min |
|----|----------|----------|--------|------------|---------|--------|-------|----------|---------|--------|-------|------------|---------|--------|-------|------------|
| 1  | 3        | 3        | 10     | 9          | 21      | 0      | 0     | 0        | 7       | 7      | 7     | 9          | 0       | 0      | 0     | 7          |
| 3  | 19       | 11       | 14     | 9          | 26      | 3      | 0     | 0        | 92      | 7      | 7     | 7          | 9       | 9      | 7     | 7          |
| 5  | 27       | 17       | 18     | 12         | 41      | 0      | 6     | 0        | 92      | 7      | 7     | 7          | 7       | 9      | 7     | 7          |
| 10 | 43       | 31       | 40     | 27         | 59      | 2      | 12    | 0        | 92      | 7      | 8     | 7          | 9       | 9      | 7     | 7          |

4.2 Performance of CDFL

Here, we report the accuracy of CDFL in comparison to SBFL and other slice-based FL techniques.

4.2.1 Overall Performance of CDFL. Table 4 contains acc@n metrics from the four studied FL techniques: CDFL, SBFL, Dynamic Slicing, and Dicing. The CDFL results are median values from ten trials, each based on CPDA computed with 20 sampled mutants (i.e., N_{mpn} = 20). CDFL and SBFL outperform the slicing-based techniques for all values of n. CDFL performs better than SBFL rankings computed with the average tiebreaker: it can locate 166.6%, 72.7%, 58.8%, and 38.7% more faults in the top 1, 3, 5, and 10 places, respectively. Compared to SBFL with the line order tiebreaker, CDFL performs better in all cases except for acc@1. Note that SBFL, which is known to require diverse test cases for better performance [40], suffers from the use of coverage adequate test suites that typically contain only one or two failing test cases. CDFL achieves better performance using the same test suite.

Since DS and Dicing use equally weighted dependence, all of their results are tied by definition. In most cases, the faulty line is executed by both passing and failing tests, which adversely affect the performance of dicing: among 92 faults, only nine faulty lines are found in the dice, all tied. SBFL is also vulnerable to ties, which can be seen from the results from minimum and maximum tiebreakers. On average, the number of tied program elements that are tied with the root cause is 20 for SBFL, and 95 for DS. In comparison, CDFL does not produce any ties.

4.2.2 Impact of Mutant Sampling on CDFL. To investigate the effect of the number of sampled mutants for each node, we vary the value of N_{mpn} from 2 to 20 by intervals of two and repeat the CDFL analysis for each configuration. Our preliminary evaluation with N_{mpn} ∈ {30, 40, 50} did not improve the accuracy of CDFL significantly beyond that obtained with N_{mpn} = 20. Note that we increase N_{mpn} by adding more samples (i.e., the set of samples with a larger N_{mpn} includes the set with a smaller N_{mpn}); this is to avoid sampling bias when considering the effect of N_{mpn}.

Figure 5 shows boxplots of CDFL results from ten trials, obtained with varying N_{mpn}, in comparison to the results of SBFL, which are shown by the dashed horizontal lines. The boxplots initially show an increasing trend as N_{mpn} increases. However, the trend tends to stabilize after N_{mpn} = 10: the difference between median of acc@n at N_{mpn} = 10 and N_{mpn} = 20 is less than, or equal to, one. The result suggests that, for the studied programs, ten mutation samples for each node is sufficient for fault localization.

4.2.3 Impact of Causal Structure on CDFL. Let us consider the impact that causal structure discovery has on the accuracy of CDFL. Table 5 presents the per-subject breakdown of acc@n results from CDFL and SBFL. CDFL produces significantly higher acc@n for

tcas, while performing much worse than SBFL for replace. For other subjects, both techniques perform similarly well.

We posit that CDFL performs better for tcas due to its simpler control-flow structure. tcas is an aircraft collision avoidance system that calculates the altitude separation [20]. The whole program is a sequence of logical formulas for the calculation of a single value, which requires minimal branching. Consequently, many computation steps are performed by both passing and failing test executions, presenting a challenge for SBFL. In comparison, CDFL considers both control and data dependency from the node trajectories, resulting in better performance.

In contrast, for many of the replace faults for which CDFL performs poorly, we find that the causal structure discovery is inadequate. By definition, a causal structure is a DAG, and cannot properly represent the cyclic dependence in a program. Breaking these cycles can lead to an incorrect set of Markovian parents, which in turn results in incorrect causal dependences. replace takes a regex pattern, a substitution string, and a target text: it matches the pattern in the target text and replaces the match with the substitution string. During its operation, replace constructs lots of cyclic dependence relations between program elements. Such a large number of the cyclic dependencies cause CPDA to struggle to model the causal dependence accurately.

Answer to RQ2: CDFL can significantly outperform other slicing based fault localisation techniques. With sufficient mutant sampling, it can outperform SBFL and does not suffer from ties.

5 DISCUSSION & FUTURE WORK

We currently use the definition of Markov parents directly in our algorithm to calculate causal dependences and the strengths of the causal connections in the CPDM. This is motivated by the fact that methods for causal discovery typically focus on observational data only, while we intervene in the program execution and thus have interventional data. Furthermore, our algorithm starts from binarized, indicator variables rather than from the raw observations. This is natural when, at least on the level of a high-level programming language, variables can contain complex structures such as trees.
However, in recent years there has been advances in learning causal structure also from interventional data and there are even algorithms for calculating optimal models with cycles [35]. Future work should investigate how to apply these advances and tools in our setting. One challenge will be to bridge the gap from a high-level program with complex values in the nodes to methods that assume numerical data only. On the other hand, there might be several benefits, such as being able to handle cyclic dependences, as well as faster, more scalable structure learning [34].

6 THREATS TO VALIDITY

Using a limited set of program inputs and approximating the dependence is a typical internal threat to any dynamic analysis, and thus also to CPDA. CPDA gets less effect from a small number of inputs through various mutations compared to dynamic slicing. RQ1 analyzes the effect of input by observing changes in CPDM when using different tests. For RQ2, we selected the tests based on the statement coverage criteria and used the same tests for other FL techniques to mitigate the threat. The sampling of mutants for CPDA poses another threat to the internal validity. To mitigate it, we sample a sufficiently large number of 100 mutants for RQ1, and conduct 10 independent trials for RQ2.

The use of the Siemens suite poses a threat to external validity. While we cannot claim that the result of RQ2 generalize, the Siemens suite is widely used in fault localization studies and enable comparison. We rely on a qualitative analysis for RQ1, and a widely studied metric (acc@n) for RQ2, to minimize the threat.

7 RELATED WORK

This paper discusses two mainstream software engineering tasks: program dependence analysis and fault localization.

7.1 Program Dependence Analysis

Static analysis attempts to uncover facts about a program that apply to any possible execution. It is therefore necessarily conservative and consequently often produces many false positives. Static dependence analysis is often used to produce a Program Dependence Graph (PDG), which was first used in compiler optimization and parallelization [14], and has subsequently found many uses [18] including program slicing [18, 19].

Dynamic dependence analysis incorporates one of more program inputs. A simple example is early dynamic slicing algorithms that computed a static slice of the PDG and then remove edges that are not executed [2]. Probabilistic Program Dependence Graph (PPDG) augments the PDG with a set of abstract states at each node that enable the use of probabilistic reasoning to analyze program behavior [6]. Later, the Bayesian Network based Program Dependence Graph (BNPDG) augments the PDG with conditional probabilities to relate the state of a node to that of its parents [44].

In comparison, CPDM is not tied to PDG, freeing us from having to solve hard data-flow problems, such as pointer analysis. Instead, we extract the dependence structure from observations of program executions. We expect to further exploit the advances in the fields of causal inference and causal discovery to refine our approach. In contrast, PPDG [6] and BNPDG [44] are tied to the lower levels of the causal hierarchy as introduced by Pearl [32] as they are based on the associative conditional probability between program states.

7.2 Fault Localization

Fault Localization (FL) aims to identify parts of the program source code that are likely to be the root cause of the observed test failures: typically FL techniques rank program elements by their relative likelihood of being the root cause [42]. One of the most widely studied FL techniques is Spectrum Based Fault Localization (SBFL), a dynamic approach that ranks program elements based on their suspicousness, which is computed from test coverage and outcomes [40]. SBFL has been widely studied both as an independent technique [1, 29, 43] and in a hybridization with other FL input features and techniques [4, 24, 27, 38]. However, it tends to produce many ties, as some program elements can share the same test coverage and outcome. Being based on coverage, SBFL also suffers from Coincidental Correctness (CC), i.e., passing executions that cover faulty elements [28].

Several existing work utilize the causal inference to fault localization. Baah et al. [5, 7] use a linear model to capture the causal effect from coverage of program elements to test outcomes. Gore et al. [16] and Shu et al. [37] apply a similar linear regression approach to predicate values and method level coverage, respectively. In comparison, CDFL focuses on general dependence relations from CPDA instead of coverage, and therefore is less affected by CC.

8 CONCLUSION

We propose CPDA that measures strengths of dependence between program elements by modeling their causal relationship. Applying causal inference on observational data, instead of using static analysis, frees dependence analysis from the burden of pointer analysis and avoids producing large number of equally weighted dependence relations. Our evaluation of CPDM, a graph representation of a program causal relations, shows that CPDA can cluster semantically related program elements, based on meaningful differences in the strengths of program dependence. We also show the utility of continuously quantifiable program dependence by using it to create a novel fault localization method (CDFL). Our empirical evaluation shows that CDFL can outperform other FL techniques even with limited test suites. Future work will consider more advanced causal modeling with an emphasis on cyclic structures, as well as other applications of CPDA.
A Direct Effect in CPDA

In this section, we reduce the formula of Definition 2.10.

**Theorem A.1.** For a subset $X$ of random nodes,

$$P(s_1, \ldots, s_n | \hat{x}) = \prod_{i \in S \cap x} P(s_i | p_{a_i})$$

for $s_1, \ldots, s_n$ consistent with $x$ otherwise.

**Theorem A.2.** Let $G = (V, E)$ be a causal structure. Given two disjoint sets of nodes, $X, Y \in E$,

$$P(y | \hat{x}) = \sum_{\{s_1, \ldots, s_m\} \cup S_x \in V} P(s_1, \ldots, s_n | \hat{x}). \quad (1)$$

**Corollary A.3.**

$$P(y | \hat{x}') = \sum_{\{s_1, \ldots, s_m\} \cup S_x \in V} P(s_1, \ldots, s_n | \hat{x}')$$

$$= \sum_{\{s_1, \ldots, s_m\} \cup S_x \in V} \left[ \sum_{s_{r_1}, \ldots, s_{r_k}} P(s_{r_1}, \ldots, s_{r_k} | \hat{x}') \right]$$

$$= \sum_{\{s_1, \ldots, s_m\} \cup S_x \in V} \sum_{s_{r_1}, \ldots, s_{r_k}} P(s_1, \ldots, s_n | \hat{x}')$$

where $\{s_{r_1}, \ldots, s_{r_k}\} \in X \cap Y^c$; since Theorem A.1,

$$= \sum_{\{s_1, \ldots, s_m\} \cup S_x \in V} \left[ \sum_{s_{r_1}, \ldots, s_{r_k}} P(s_1, \ldots, s_{r_k}, \ldots, s_n | \hat{x}') \right]$$

$$= \cdots = \sum_{\{s_1, \ldots, s_m\} \cup S_x \in V} P(s_1, \ldots, s_n | \hat{x}')$$

**Corollary A.4.** For a statement $Y \in V$, using Corollary A.3,

$$P(y | \hat{p}_a Y) = \sum_{\{s_1, \ldots, s_m\} \cup S_y \cup \hat{p}_a Y} P(s_1, \ldots, s_n | \hat{p}_a Y)$$

$$= \sum_{\{s_1, \ldots, s_m\} \cup S_y \cup \hat{p}_a Y} \left[ \prod_{j \in S_y \cup \hat{p}_a Y} P(s_j | p_{a_j}) \right] \quad \text{(by Theorem A.1)}$$

$$= P(y | \hat{p}_a Y) = \sum_{\{s_1, \ldots, s_m\} \cup S_y \cup \hat{p}_a Y} \left[ \prod_{j \in S_y \cup \hat{p}_a Y} P(s_j | p_{a_j}) \right]$$

$$= P(y | \hat{p}_a Y),$$

since $\sum_{\{s_1, \ldots, s_m\} \cup S_y \cup \hat{p}_a Y} \left[ \prod_{j \in S_y \cup \hat{p}_a Y} P(s_j | p_{a_j}) \right] = 1.$

**Corollary A.5.**

$$DD(S_i, S_j)$$

$$= \frac{1}{|I|} \sum_{i \in I} \text{NDE}_{o_i, S_i} (S_j)$$

$$= \frac{1}{|I|} \sum_{i \in I} \sum_{z} \left[ \left\{ E_{o_i} (S_j | do(S_i = 1, z) - E_{o_i} (S_j | do(S_i = 0, z)) \right\} \right.$$}

$$\times P_{o_i} (z | do(S_i = 0))$$

$$= \frac{1}{|I|} \sum_{i \in I} \sum_{z} \left[ \left\{ P_{o_i} (S_j = 1 | do(S_i = 1, z)) - P_{o_i} (S_j = 1 | do(S_i = 0, z)) \right\} \right.$$}

$$\times P_{o_i} (z | do(S_i = 0)) \right),$$

where $Z = PA_j \setminus S_i$. Since $\{S_i \cup Z = PA_j\}$, by Corollary A.4,

$$= \frac{1}{|I|} \sum_{i \in I} \sum_{z} \left[ \left\{ P_{o_i} (S_j = 1 | S_i = 1, z) - P_{o_i} (S_j = 1 | S_i = 0, z) \right\} \right.$$}

$$\times P_{o_i} (z | do(S_i = 0)) \right).$$

Here, by Corollary 2.7,

$$P_{o_i} (z | do(S_i = 0)) = \sum_{p_{a_i}} P_{o_i} (z | S_i = 0, p_{a_i}) P_{o_i} (p_{a_i}).$$

12