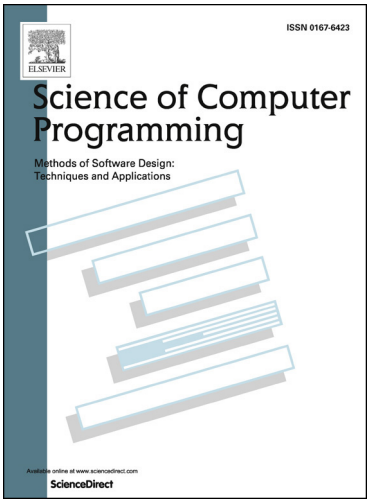
Journal Pre-proof



Causal Program Dependence Analysis

Seongmin Lee, Dave Binkley, Robert Feldt, Nicolas Gold and Shin Yoo

PII: S0167-6423(24)00131-X
DOI: https://doi.org/10.1016/j.scico.2024.103208
Reference: SCICO 103208

To appear in: *Science of Computer Programming*

Received date: 11 March 2024
Revised date: 21 August 2024
Accepted date: 8 September 2024

Please cite this article as: S. Lee, D. Binkley, R. Feldt et al., Causal Program Dependence Analysis, *Science of Computer Programming*, 103208, doi: https://doi.org/10.1016/j.scico.2024.103208.

This is a PDF file of an article that has undergone enhancements after acceptance, such as the addition of a cover page and metadata, and formatting for readability, but it is not yet the definitive version of record. This version will undergo additional copyediting, typesetting and review before it is published in its final form, but we are providing this version to give early visibility of the article. Please note that, during the production process, errors may be discovered which could affect the content, and all legal disclaimers that apply to the journal pertain.

© 2024 Published by Elsevier.

Journal Pre-proof

Causal Program Dependence Analysis

Seongmin Lee[a], Dave Binkley[b], Robert Feldt[c], Nicolas Gold[d], Shin Yoo[e]

[a]*Max Planck Institute for Security and Privacy, 140 Universitätsstr., Bochum, Germany44799,*
[b]*Loyola University Maryland, 4501 North Charles Street, Baltimore, USAMD 21210,*
[c]*Chalmers University of Technology, Chalmersplatsen 4, Göteborg, Sweden412 96,*
[d]*University College London, Gower St, London, UKWC1E 6BT,*
[e]*Korea Advanced Institute of Science and Technology, 291 Daehak Ro, Yuseong Gu, Daejeon, Republic of Korea34141,*

Abstract

Discovering how program components affect one another plays a fundamental role in aiding engineers comprehend and maintain a software system. Despite the fact that the degree to which one program component depends upon another can vary in strength, traditional dependence analysis typically ignores such nuance. To account for this nuance in dependence-based analysis, we propose Causal Program Dependence Analysis (CPDA), a framework based on causal inference that captures the degree (or strength) of the dependence between program elements. For a given program, CPDA intervenes in the program execution to observe changes in value at selected points in the source code. It observes the association between program elements by constructing and executing modified versions of a program (requiring only light-weight parsing rather than sophisticated static analysis). CPDA applies causal inference to the observed changes to identify and estimate the strength of the dependence relations between program elements. We explore the advantages of CPDA's quantified dependence by presenting results for several applications. Our further qualitative evaluation demonstrates 1) that observing different levels of dependence facilitates grouping various functional aspects found in a program and 2) how focusing on the relative strength of the dependences for a particular program element provides a detailed context for that element. Furthermore, a case study that applies CPDA to debugging illustrates how it can improve engineer productivity.

Keywords: CPDA, dependency analysis, causal inference, observation-based analysis

Email addresses: seongmin.lee@mpi-sp.org (Seongmin Lee), binkley@cs.loyola.edu (Dave Binkley), robert.feldt@chalmers.se (Robert Feldt), n.gold@ucl.ac.uk (Nicolas Gold), shin.yoo@kaist.ac.kr (Shin Yoo)

Journal Pre-proof

Journal Pre-proof

1. Introduction

Program dependence analysis is fundamental to understanding the semantics of a program (Horwitz and Reps, 1992). When working with the code, it provides a useful lens to reduce the number of program elements that must be considered for a wide range of tasks, such as program comprehension (Zhifeng Yu and Rajlich, 2001), software testing (Binkley, 1997), debugging (Jiang et al., 2017; Lee, 2020), refactoring (Ettinger and Verbaere, 2004), maintenance (Gallagher and Lyle, 1991), and security (Karim et al., 2018).

In a program, each component depends on some number of other program components. With its roots in compiler optimization, traditional static dependence analysis attempts to safely approximate the set of dependence relationships with respect to *all* possible executions of the program resulting in a set of *binary* relations: there either is, or is not, a dependence between any two program components. Even in this simplified binary setting, static dependence analysis often becomes quite involved, for example when coping with the nuances of program language semantics. A good example is pointer analysis, which is not only computationally expensive[1] but also prone to producing a large number of false-positives (Ramalingam, 1994; Lee et al., 2019b). These false-positives often degrade the usefulness of the analysis in downstream tools.

Furthermore, traditional dependence analysis does not need to determine the *relative strength* of a dependence. However, some dependences are stronger than others. Consider two uses of a variable where one rarely has an impact on the computation of a third but the second always does. For example, the use of `a` in the statement `b = (a == 42)` rarely changes the value assigned to `b`. In contrast, the use in `b = a + 42` always changes the value of `b`. Despite this difference, in both cases, there exists a dependence from `a`'s definition to its use in the assignment to `b`.

A few dynamic approaches have aimed to model dependence strength. For example, the PPDG (Baah et al., 2010) and BNPDG (Yu et al., 2017) estimate the frequency of a data flow by measuring the conditional probability of a particular set of reaching predecessors of a program element given the state at each predecessor during execution on a set of test cases. This notion of dependence strength was shown to successfully aid in fault localization. However, a drawback of these two approaches is that they build on top of static dependence analysis; thus, they inherit its limitations causing their performance to be limited by the quality of the underlying static analysis. A more recent approach with similar goals, MOAD (Lee et al., 2021), avoids the expense of static analysis by using dynamic observation-based analysis. While the resulting technique can capture the effect of one program element on another it is unable to reason about how one program element affects another through a chain of cause-and-effect relationships.

[1] For example, Andersen's well-known flow-insensitive, context-insensitive algorithm has an $O(n^3)$ running time. In the more general case, the complexity becomes exponential.

Journal Pre-proof

This paper introduces *Causal Program Dependence Analysis (CPDA)*, a framework that uncovers *causal dependence* between program elements. This includes the ability to capture the strength of the dependences between program elements. Given a set of program executions, CPDA discovers a *causal structure* that indicates the direct cause-and-effect relationships between program elements. This structure is based on the observed behavior of a set of elements from the program. By applying techniques from causal inference (Pearl, 2009; Pearl et al., 2009) over this causal structure, CPDA produces two measures of program dependence strength using two metrics from the causal inference literature. *Natural direct effect (NDE)* captures the effect of one program element on another *excluding* effects that pass through other elements (Pearl, 2001). *Average causal effect (ACE)* takes into account both NDE's direct effect and indirect effects (Holland, 1988). Thus, ACE captures the total effect that a change to a program element has on the behavior of other program elements. Note that the dependence measured by CPDA is relative to a set of reference executions (i.e., the executions using a particular test suite) that are used to observe the program behavior. Whilst this means that the quality of the analysis is dependent on the particular executions used, such an approach is widespread and has been successfully used in dynamic analysis (Lee and Böhme, 2023; Ernst et al., 2001; Binkley et al., 2015). The benefits of having execution-specific dependences are explored in Sections 9.2.2 and 9.2.3.

We also propose a novel way to visualize program dependence using the *Causal Program Dependence Model (CPDM)*. This weighted dependence graph shows the causal structure of a program, annotated with NDEs. The paper describes how the CPDM can be used to reason about program dependence more precisely and more intricately than its predecessors. We conjecture that our two measures of quantitative dependence and the CPDM form a viable new foundation for a range of program analysis techniques.

As mentioned above, CPDA has the advantage that it avoids the need for computationally expensive static analysis. In addition, it does not require the troublesome overhead of coordinating static analyses, for example, adjusting the right level of abstraction for the program or the analysis, something that is typically complex and requires a lot of manual effort and domain knowledge, and is unavoidable to make static analysis accurate. It requires only light-weight parsing for the instrumentation. Specifically, points of interest in the code are modified so that we can introduce simple mutations to the state and then observe their effect on subsequent computations. An additional benefit of this approach is that it can directly model dependences that go outside the formal semantics. For example, CPDA can capture dependences caused by values that get transferred through a database or through the file system. Furthermore, it can be applied to heterogeneous systems built using multiple programming languages or system making use of third-party binary libraries provided that the part of the program of interest can by instrumented. The benefit of this kind of observational approach has been demonstrated in the work on Observation-based Slicing (Binkley et al., 2014; Lee et al., 2020).

Our final contribution comes in the form of two new causal structure discov-

Journal Pre-proof

ery algorithms. Existing algorithms consider graphs with fewer than a hundred nodes (Singh and Valtorta, 1995; Scutari et al., 2019; Rantanen et al., 2018, 2020b,a), which is not sufficient for large programs. Our two new algorithms for causal structure discovery are designed to capture different aspects of the qualification of causal relations. The first, the *Conditional Probability* (*CP*) method considers probabilistic aspects, while the second, the *Hitting Set* (*HS*) method captures a discrete/deterministic aspect. We empirically evaluate the causal structures discovered by the two algorithms and consider the extent to which each describes the dependence relations in a program by comparing the two with a ground-truth program dependence graph. We also explore the range of advantages of CPDA's quantified program dependence by presenting results of its application to three software engineering applications. For example, clustering the nodes and observing the dependences in the CPDM facilitates the uncovering of functional aspects of the code. As a second example, focusing on the relative strength of the dependences to and from a particular program element helps bring out semantic relations between the element and the code around it. Finally, we explore CPDA's application in downstream tasks through a debugging case study.

The main contributions of this work include the following:

- We propose Causal Program Dependence Analysis (CPDA), a dependence analysis framework based on causal inference that supports the quantification of dependence strength between program components.
- We propose two structure discovery algorithms with considerably better scalability. These two discover the causal structure of the dependence relations in a program. We empirically evaluate the two both quantitatively and qualitatively.
- We visualize and reason about the result of a causal analysis using our Causal Program Dependence Model (CPDM), a graph-structured program dependence model that provides a rich representation of quantified dependence.
- We consider several related applications and examine the advantages and limits of quantified program dependence via empirical experiments.

The rest of this paper is organized as follows. We first present a motivating example describing various characteristics of the program dependence found in the program semantics in Section 2. Then Section 3 explains causal inference, which is the theoretical background of CPDA. After presenting the procedure of CPDA in Section 4, we explain its details starting with the probabilistic representation of program semantics (Sec. 5), the structure discovery algorithms (Sec. 6), and the measures of quantified dependence (Sec. 7). In Section 8 we describe our experimental configuration. Then in Section 9, we first assess the accuracy and efficiency of the structure discovery algorithms. Then, we investigate the characteristics and potential of CPDA's quantified dependence using

Journal Pre-proof

several applications. Threats to validity are discussed in Section 10. Finally, we review related work in Section 11, discuss future work in Section 12 and conclude in Section 13.

2. Illustrative Example

```
1 a = 42;
2 pred = input(); // true or false
3 b = a + 1;
4 if (pred) {
5     c = 2 * a + b % 2;
6     d = c - 1;
7 }
8 e = 3;
```

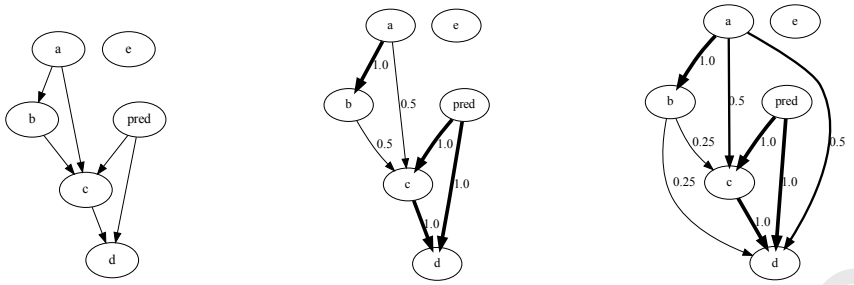
Figure 1: Code of the motivating example.

To help motivate our approach, we present an illustrative example that demonstrates the limitations of existing program dependence analysis techniques. Figure 1 is an example program showing various characteristics of program dependence. The value in variable `b` in line 3 is the value in variable `a` plus one. Therefore, a value change of variable `a` always affects the value in variable `b`; in terms of how often one affects another, variable `a` *strongly* affects variable `b`. Similarly, variables `c` and `d` in lines 5 and 6 are only assigned if the predicate `pred` is `true`; thus, `c` and `d` strongly depend on `pred`. On the other hand, the value in variable `c` is affected by the change of the value in variable `b` only when `pred` is `true` and the `b`'s parity changes (due to the remainder operator '`%`'). Assuming the domain of `b` is an integer, only in half of the instances does variable `b` affect variable `c`; thus, `c` *weakly* depends on `b` compared to `a`. Another property of dependence is *immediacy*. For instance, variable `b` affects both values of variable `c` and `d`; while `b` *directly* affects `c`'s value, it *indirectly* affects `d`'s value through the value of `c`. Variable `a` in line 1 affects variable `c` in line 5 both directly and indirectly (through the value of `b`).

Static analysis is capable of differentiating the direct and indirect dependence using the formal semantics of the programming language. However, it does not distinguish the magnitude of the dependence. In the static dependence model, the dependency from variable `a` to variable `b` and variable `b` to variable `c` is the same: "yes, there is dependence." The nature of the static analysis, trying to find all possible dependences, brings too many dependences in a large program because of both theoretical (Ramalingam, 1994) and practical reasons (Binkley et al., 2015). Without any way to discriminate those dependences, this large number of dependences does not help the user understand the program and hinders subsequent analysis. Figure 2a shows the dependence model from static analysis.

The Probabilistic Program Dependence Graph (PPDG) (Baah et al., 2010) and the Bayesian Network-based Program Dependence Graph (BNPDG) (Yu

Journal Pre-proof



(a) Static analysis (b) PPDG & BNPDG (c) MOAD

Figure 2: Dependence model using existing dependence analysis methods.

et al., 2016) are recent dynamic dependence analyses that model dependence strength based on the frequency of def-use chains. For each program element, the PPDG models the conditional probability of a particular set of reaching predecessors given the state at each predecessor during execution. The PPDG estimates conditional probability as a relative frequency observed in a set of node-state traces from a set of test cases. Later, the BNPDG reformed the structure of PPDG to a DAG in order to regard it as a Bayesian Network and compute the conditional probability between non-adjacent nodes (Yu et al., 2017).

While PPDG and BNPDG consider the conditional probability of a state of a program element given the states of its predecessors, an effect from a single predecessor to the program element can be derived by marginalizing the conditional probability over the other predecessors. Figure 2b shows the conceptual dependence model of the PPDG and the BNPDG using marginalization. While both models assign 0.5 to the dependence from `a` and `b` to `c` as `a` and `b` only affect `c` when `pred` is True they cannot discriminate the degree of dependence between `a` and `b` from `c`. This is because the PPDG and the BNPDG are incapable of noticing the value change of variables.

Our previous work introduced MOAD (Lee et al., 2021), an observation-based dependence model capable of identifying the value change of variables in response to deletion mutations. Given a program, it removes various combinations of program elements from the source code and observes the consequence of each deletion. Given those observations, MOAD employs several statistical methods to approximate the degree of dependence between program elements. The main hypothesis of statistical methods is that the degree of A affecting B is proportional to the conditional probability of the change in B's behavior, given A is mutated. As shown in Figure 2c, MOAD distinguishes the difference in the degree of dependence between `a` and `b` from `c`. However, MOAD is incapable of identifying the dependence structure (i.e., whether the effect is direct or indirect). It only measures the strength of dependence, ignoring whether one

Journal Pre-proof

affects another directly or indirectly. Distinguishing direct and indirect effects is important for making the analysis *path-sensitive*, enhancing the understanding of edge-wise and path-wise information flow. By identifying whether an effect is direct or indirect, one can more accurately trace causal influence through different edges and paths. This distinction is particularly valuable in analyses like debugging or security assessments. For instance, if a vulnerability, such as an information leakage, arises on a specific execution path, knowing the extent of the causal effect along that path—as opposed to others—becomes crucial. Thus, differentiating direct from indirect effects greatly improves our ability to pinpoint and address issues in complex systems.

In this work, we present a dependence analysis technique that discovers both the structure and the degree of the dependence. This is similar to how causal analysis extends the statistical analysis to study the actual causal structure as well as estimate the strength of the dependences. In Section 4, we introduce *Causal Program Dependence Analysis*, a novel dependence analysis utilizing causal inference to address the above-mentioned limitations.

3. Background

This section introduces causal inference, the fundamental methodology that CPDA uses to identify and quantify program dependence.

3.1. Causal Inference

Causal inference is a mathematical theory for analyzing which events cause one or more effects (Pearl, 2009; Pearl et al., 2009). Increasingly, it provides practical tools for data analysis that can be used instead of, or together with, existing statistical and other data analysis methods (Peters et al., 2017). While statistical methods focus on identifying and modeling associations, the causal analysis adds ways of studying which events actually precede and thus lead to (cause) other events. It further determines how large these effects are. The distinctions involved are clarified by the so-called 'ladder of causality' where classical, associative, statistical analysis ('what happens together?') is the ground level, the analysis of interventions ('what happens if we do this?') one step up, and counterfactual analysis ('what if something else, that didn't happen, will (or had) happen(ed)?') the top level (Pearl, 2019).

Theoretically, the benefits of causal inference are clear, e.g., it is aligned with the ultimate goals of science in attempting to explain the reasons for observed phenomena. However, a causal analysis can also provide more direct and practical benefits and ultimately lead to more robust models and better decisions than those derived from non-causal statistical analyses. As an example, in medicine, a re-analysis of data on hip fractures among the elderly found that the causal analysis was able to identify which events mediate the effect of the others and to what extent, in addition to providing predictions on par with traditional methods (Caillet et al., 2015). In another study, Richens et al. (2020) showed that medical diagnosis based on causal inference performed almost twice as well

Journal Pre-proof

(25^{th} percentile vs. 48^{th} of the performance of human doctors) than classical, associative/statistical methods.

3.2. Causal Model and Causal Structure

A *probabilistic model* (also called a probability space) is a mathematical model encoding the association information between events. It consists of three components: 1) a sample space, which is the set of all possible outcomes, 2) an event space, which is a set of events, where an event is a subset of a sample space, and 3) a joint probability distribution over the random variables of the events[2]. Causal inference can elevate the probabilistic model to demonstrate the causal relationship between the events.

A key element of modern-day causal inference is its use of directed acyclic graphs (DAGs) to model the dependence structure between the random variables of different events in a probabilistic model. The resulting model of the causal inference, called the *causal model*, consists of the *causal structure*, which is the DAG of the dependence structure, and the mathematical relations between the random variables (Pearl, 2009). Nodes in the DAG denote random variables, and edges denote how the values of the random variables cause changes in the values of other random variables. To be specific, the direct predecessors (parents) of the node (child) in the DAG are a minimal set of nodes directly affecting the node; therefore, the parents screen the child from the effect of any other nodes which may indirectly affect the child.

The mathematical relations in the causal model indicate how to estimate the value of a child random variable based on the values of their parent(s). There are different types of causal models that differ in how the relations are specified. In a so-called *structural equation model* (Pearl, 2009), this is achieved via equations that relate each child node to the nodes from which it has incoming DAG edges. The equations are typically linear but more general forms can be used. Another model variant, the *probabilistic causal model*, factorizes the probability distribution over the random variables into a set of independent conditional probability distributions, each denoting the atomic cause-and-effect relations between variables (Pearl, 2009). The structural equation model works well if the relation between factors is expected to have a certain pattern (e.g., linear), while the probabilistic causal model can model a more generic (non-parametric) relation. Therefore, in this paper, we use the probabilistic causal model to model the program dependence. The (probabilistic) causal model satisfies the *local Markov condition*:

Definition 3.1 (Local Markov Condition). *Let $G = (V, E)$ be a direct acyclic graph (DAG) and let P_V be a probability distribution over the nodes V of G. G and P_V satisfy the local Markov condition if every node in V is conditionally independent of its non-descendants, given its parents.*

[2]To be specific, it is a probability measure on events, which assigns a probability to each event (Stroock, 2010). Refer to ((Pearl, 2009), Chapter 1.1.2) for more details on probability spaces and its context in causal inference.

Journal Pre-proof

For instance, let us consider a situation where whether it is dawn and there is dew on the grass (A), and whether it is raining (B) are the two factors that affect whether the grass is slippery (C), which decides whether a person who is walking on the grass will slip and fall (D); let the probability distribution $P_{A..D}$ describes the situation. Then, regarding $P_{A..D}$, DAG $G \triangleq \{A \to C,\ B \to C,\ C \to D\}$ meets the local Markov condition, but DAG $G' \triangleq \{A \to C,\ C \to D\}$ does not because C is not conditionally independent of A given B, which means that G' does not express that the grass can be slippery because of the rain. Nonetheless, none of A and/or B make C conditionally independent of D, the descendent of C, because D is the result of C.

For each random variable (node) in the probability distribution, a minimal set of predecessors that satisfies the local Markov condition with the given random variable is called *Markovian parents*. A conventional approach to construct the causal structure is thus to find the Markovian parents of each node and draw edges from every Markovian parent to the node. While the causal structure, i.e., the DAG, is sometimes known or can be formulated based on some external theory, one can also use so-called structural learning (also known as causal discovery) to identify the causal structure by searching for the DAG that satisfies the local Markov condition from the data (Spirtes and Zhang, 2016).

3.3. Causal Effect

A hallmark of causal inference is that its DAGs can be used to guide which (random) variables to intervene on (i.e., change) to calculate the effects of one variable on another, given the observations. The *causal effect*, the measured effect by causal inference, distinguishes itself from the conditional probability, which is a measure representing the association rather than the causation. The conditional probability $P(\cdot|x)$ represents the probability when one *observes* that X has value x (we use the lower case symbol x to denote a particular observed value of the corresponding uppercase variable X). What this illustrates is an association between $X = x$ and other events. On the other hand, the *causal effect*, denoted as $P(\cdot|do(x))$, is a probability when we *force* X to have the value x. The difference between "forcing X to x" and "observing that X is x" concerns whether there is any actual effect of X, or, somehow, there is a correlation with X either by X itself or by other variables.

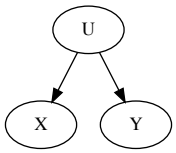
Before getting onto the formal definition of the causal effect, we first informally introduce the notion by way of an example. Consider the situation shown in Figure 3, where sleeping with shoes on (denoted the event as X) often occurs together with having a headache (denoted as Y) the next morning when there was heavy drinking last night (denoted as U), i.e., having a headache when someone slept with shoes on has a higher chance than the chance of normally having a headache ($P(\mathrm{Y} \mid \mathrm{X}) \gtrsim P(\mathrm{Y})$). However, since sleeping with shoes is not the cause of the headache, putting shoes on a sleeping person, i.e., $do(\mathrm{X})$, will not increase the chance of a headache ($P(\mathrm{Y} \mid do(\mathrm{X})) \approx P(\mathrm{Y})$).

By having a causal structure, we can estimate the causal effect by controlling the *confounding bias*. A confounding bias is a distortion representing the event that is associated with, but not causally related to, the observation induced

Journal Pre-proof

(a)

U : drink last night
X : sleep with shoes on
Y : have a headache



(b)

	U		¬U	
	X	¬X	X	¬X
Y	100	0	1	9
¬Y	0	0	9	81

(c)

$$
\begin{aligned}
P(\mathrm{Y} \mid \mathrm{X}) &= 101/110 \approx 0.92 \\
P(\mathrm{Y} \mid do(\mathrm{X})) &= P(\mathrm{Y} \mid \mathrm{X}, \mathrm{U}) \cdot P(\mathrm{U}) + P(\mathrm{Y} \mid \mathrm{X}, \neg \mathrm{U}) \cdot P(\neg \mathrm{U}) \\
&= 100/100 \times 100/200 + 1/10 \times 100/200 \\
&= 0.55 = P(\mathrm{Y})
\end{aligned}
$$

Figure 3: An example of 200 observations showing the difference between conditional probability and causal effect: (a) is the causal structure, (b) is the table of occurrence of the situations, and (c) shows the computation of the conditional probability and the causal effect.

by the common cause (U in the previous example), which appears through the so-called *backdoor path* (in the case of Figure 3, X $\leftarrow$ U $\rightarrow$ Y) in the causal DAG (Pearl, 2009). 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.

Based on Pearl (2009), the causal effect $P(\cdot|do(x))$ is formally defined as follows:

Definition 3.2 (Causal Effect). *Let $G = (V, E)$ be a causal structure. Given two disjoint sets of nodes, $X, Y \subset V$, the causal effect of X on Y, denoted as $P(y \mid do(x))$, is a function from X to the space of probability distributions on Y. For each observed value x of X, $P(y \mid do(x))$ gives the probability that $Y = y$ is induced by deleting from the causal structure the edges to the nodes in X and substituting $X = x$. The causal effect $P(y \mid do(x))$ is calculated as follows:*

$$P(y \mid do(x)) = \sum_{mp_X \in \mathrm{dom}(MP_X)} P(y \mid x, mp_X) P(mp_X)\,,$$

where mp_X represents the particular observed set of states of MP_X, the set of Markovian parents of X.

In Figure 3, while the conditional probability of having a headache (Y) when sleeping with shoes on (X), $P(\mathrm{Y} \mid \mathrm{X})$, is high, showing heavy association, based on the computation of Definition 3.2, the causal effect of sleeping with shoes to having a headache, $P(\mathrm{Y} \mid do(\mathrm{X}))$, is the same as the probability of having a headache. In our example, there is thus no causal effect from sleeping with shoes on to have a headache, as one would expect.

Journal Pre-proof

4. Overview of Causal Program Dependence Analysis

Causal Program Dependence Analysis (CPDA) aims to model and quantify the strength of program dependence relations. We define the *behavior* of a program element as the state or the value it takes during program execution (a detailed definition is discussed in Section 5.1), and we interpret the *dependency* between program elements as the effect of a change in the behavior of one program element on the behavior of another program element. A different meaning of the term 'effect' can be considered in the context of quantifying the strength of program dependence relations, for instance, the magnitude of the change in the behavior of program elements. In this initial work of CPDA, we focus on the likelihood/probability of a change in the behavior. A discussion of the extension of CPDA to different notions of effect is discussed in Section 12.

CPDA models the dependences between program elements using causal inference. Thus, the dependences reported by CPDA are not binary: rather, they represent *how likely* a change to the value of a program element S_i is to *cause* a change to the value of another element S_j.

Definition 4.1 (Causal Dependency (Abstract)). *Given a Program $\mathcal{P}$, let S_i and S_j be program elements in $\mathcal{P}$. The **causal dependency** from S_i to S_j is the probability that a change in the value of S_i will cause a change in the value of S_j.*

In order to model the program dependency using causal inference, we initially define a probabilistic model over the runtime behavior of the program elements. A sample from the probabilistic model, denoted as the Δ-*execution model*, represents a set of program elements whose behaviors change together during execution. To get a sample from the Δ-execution model, we first capture the behavior of the original program when executed on a test suite. These executions are used as an oracle. We then observe[3] which program elements behave differently in the mutated program execution. Observations of which program elements behave differently together are used as input data for the causal inference. The output of the causal inference is a set of causal dependences between program elements.

Figure 4 shows the overall framework of CPDA. Given a program, CPDA identifies target program elements to analyze, and subsequently instruments the code (Sec. 5.2). The instrumentation allows us to generate an *oracle*, a

[3]The term "observation" used in our work differs from how "observation" is generally used (in contrast to "intervention") in causal inference. In a causal inference study, one naturally gets observational data from the event space of interest, while one needs to take actions that lead to changes to the event space to get interventional data. In our work, we use the term "observation", with its more general meaning, for (a part of) the observable behavior of program elements during execution regardless of whether it arises from the original or the mutated programs. The causal inference in our work thus uses both purely observational data (the behaviors of the unmutated program execution) as well as interventional data (the behaviors of the mutated program executions).

Journal Pre-proof

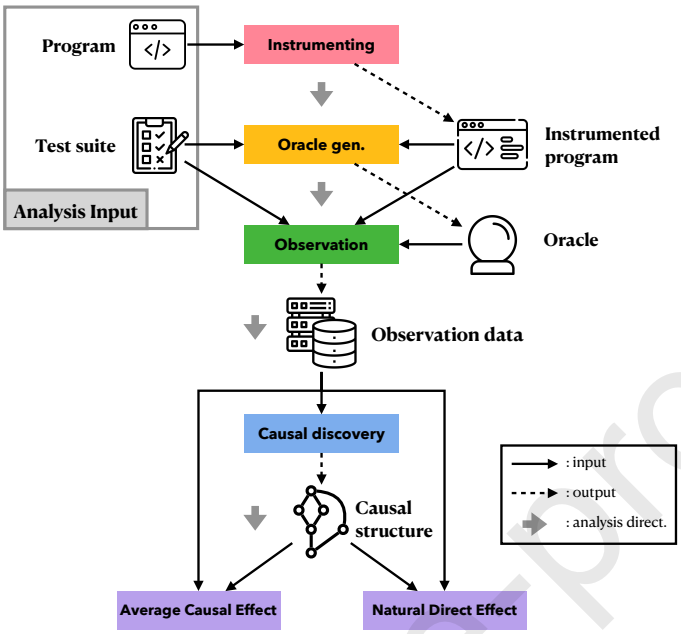


Figure 4: Framework of CPDA

set of recorded behaviors, i.e., states of program elements from the original, unmutated program. Mutation then allows us to apply interventions to the state of a program element at runtime and monitor the resulting changes (Untch et al., 1993). Using these observations, CPDA first builds a *causal structure* of the program (Sec. 6). Then, using this causal structure and the observation data, CPDA performs causal inference to calculate two dependence measures between program elements (Sec. 7). First, *average causal effect* measures the total effect from one program element to another, including both direct and indirect effect, while the second, *natural direct effect*, measures only the direct effect. Finally, by annotating the natural direct effect on the edges of the causal structure, CPDA produces a Causal Program Dependence Model (CPDM), a graphical representation of the quantified program dependence.

5. Transforming Program Semantics to a Probability Distribution

5.1. Δ-execution model and Probability of a Behavior Change

Given a probabilistic model that describes how the events are associated, causal inference infers the causal relationship between the events. Hence, to infer causal dependency, we need to define an appropriate *probabilistic model* over which the causal inference can be performed. The elements of the probabilistic model are *program elements*, which capture a program variable at a specific location in the source code. We design a probabilistic model that describes which

Journal Pre-proof

program elements are dynamically associated, so that their behaviors change together during execution, i.e., during a single run of the program from the test suite. Consequently, the causality derived from the probabilistic model represents which program element's behavioral change is likely to cause another program element's behavioral change, i.e., the causal dependency in Definition 4.1. Specifically, we define a probabilistic model where each sample in the space corresponds to a set of program elements whose behavior changes in the intervened execution, i.e., the execution with a mutated program. We call this probabilistic model a Δ-*execution model*.

We next formalize the behavior of a program as the behavior of its program elements. The behavior of a program element is the *trajectory* (the sequence of values) the corresponding program variable takes on at a specific location in the source code during execution. Given a Δ-execution model of a program, a *behavior change* in a program element is defined as follows:

Definition 5.1 (Behavior change of a program element). *The behavior of program element $S_i = \langle v, l \rangle$, where v is a variable and l is a location in the source code, in Program $\mathcal{P}$ has changed for a given mutation and input if the trajectory for S_i in the mutated program $\mathcal{P}'$ is different from the trajectory of S_i using $\mathcal{P}$.*

Given a Δ-execution model of a program, CPDA infers a *causal model*. Each node in CPDA's causal model is a random variable representing whether the behavior of a program element has changed under $\mathcal{P}'$ or not. Henceforth, we overload the notation S_i to represent both the program element itself and the random variable, denoting whether the behavior of S_i has changed under $\mathcal{P}'$ or not; $S_i = 1$ if changed, 0 otherwise. Consequently, the causal structure of CPDA's causal model identifies the dependency between program elements, and the *(average) causal effect*, a metric of causation between two nodes in CPDA's causal model, measures the quantity of the causal dependency.

We sample *observations* from the Δ-execution model through the following procedure. First, we run the program with a given test suite and capture the original behavior as the trajectory of each program element. The captured behavior serves as an oracle of the original behavior of the program. After we produce the oracle, we repeat the execution while selectively mutating the value of each program element one at a time. Based on these executions, we identify behavior changes in program elements due to the mutation by comparing the trajectory of each program element after the mutation with the corresponding trajectory in the oracle. For each mutated execution, we get a single observation that is a boolean vector whose length is equal to the number of program elements in the program: each value in the observation indicates whether the behavior of the corresponding program element has changed or not. Given a set of observations, we can calculate the probability of a behavior change in a program element as follows:

Definition 5.2 (Probability of a behavior change). *For a program element S_i from $\mathcal{P}$ and a set of observations O, $P_O(S_i = 1)$ represents the probability of*

Journal Pre-proof

a behavior change in S_i when not mutated,

$$P_O(S_i = 1) = \frac{|\{obs \in O^* \mid obs[i] = 1\}|}{|O^*|} = 1 - P_O(S_i = 0)\,,$$

where $O^* = \{o \mid o \in O \wedge o$ is not generated by mutating $S_i\}$.

Moreover, we calculate the *conditional* probability of a behavior change:

Definition 5.3 (Conditional probability of a behavior change). *For program elements S_i and S_j from $\mathcal{P}$ and a set of observations O, $P_O(S_i = 1 \mid S_j = 1)$ represents the probability of a behavior change in S_i when not mutated, given a behavior change in S_j,*

$$\begin{aligned} P_O(S_i = 1 \mid S_j = 1) &= \frac{P_O(S_i = 1 \wedge S_j = 1)}{P_O(S_j = 1)} \\ &= \frac{|\{obs \in O^* \mid obs[i] = 1 \wedge obs[j] = 1\}|}{|\{obs \in O^* \mid obs[j] = 1\}|}\,, \end{aligned}$$

where $O^* = \{o \mid o \in O \wedge o$ is not generated by mutating $S_i\}$.

The conditional probability in Definition 5.3 becomes the foundation of the causal dependences that we calculate later. The use of O^* excludes the mutation at S_i itself to avoid negating the effect from the behavior change of S_j with a mutation of S_i itself.

5.2. Implementation

We instrument the target program based on the set of program elements whose causal dependence we seek to analyze; the analysis result is thus subject to the program elements chosen. Our instrumentation allows us to intervene in the *runtime behavior* of the program by mutating a program variable's value during execution. It also allows us to observe the trajectory of the program elements, either from the original execution if no intervention is applied or from the mutated execution if an intervention is applied; the difference in the trajectory of a program element between the original and mutated executions indicates that the mutation has changed the behavior of the program element. In the remainder of the paper, we consider the following as program elements: a left-hand side program variable of an assignment statement, a function parameter, a predicate expression, and a return expression.

To efficiently produce a large and diverse set of observations for Program $\mathcal{P}$, we construct a *super mutant* (Untch et al., 1993), i.e., a meta-mutated program that takes as input a mutation position (the unique index of a program element) and a mutation value that will be used whenever the program element is executed. This approach reduces the number of compilations required to support multiple mutations. The instrumentation begins by indexing all the program elements in the target program using a parser and injecting a helper function.

Journal Pre-proof

```
[Running the instrumented program]
 $ ./inst_prog <MUT_IDX> <MUT_VAL> <input>

[Helper function OBS]
 def OBS(node_idx, var_name, val) {
   if (node_idx == MUT_IDX) {
     print("OBS", node_idx, var_name, MUT_VAL)
     return MUT_VAL
   }
   else {
     print("OBS", node_idx, var_name, val)
     return val
   }
 }

[Add the helper function for each program element]
 foo = 3; foo = OBS(42, "foo", foo)
```

Figure 5: An example of the trajectory logging instrumentation and the helper function.

Figure 5 shows an example of the helper function, how it is injected after each program element, and how the instrumented program is executed with the target mutation index and the mutation value. The helper function, when given a target program element index (`MUT_IDX`), will overwrite (i.e., mutate) the value of the program element to the given mutation value (`MUT_VAL`) and log the result of mutation during execution. For all the other program elements, the helper function simply logs the current value observed during execution (this records each program element's trajectory).

5.2.1. Oracle

After instrumentation, we can obtain the oracle trajectory for each program element using the given test suite. An oracle trajectory for a program element is simply a collection of all of its trajectories, one per test input from the test suite when no mutation is applied during execution.

5.2.2. Mutation

CPDA currently targets variables of the following primative types: `bool`, `char`, `int`, `long`, `float`, `double`, and `string`. We propose a mutation strategy for each type as follows. First, if domain knowledge clearly specifies the range of possible values (e.g., an enumerated type), we simply sample from the given range with uniform probability. Otherwise, we aim to choose the diverse mutation values but, simultaneously, values that the corresponding program element may have, which are close to the observed values in the oracle. For Booleans, the mutation simply negates the original value. For scalar types, we first build a Gaussian distribution whose mean and standard deviation are the sample mean

Journal Pre-proof

and sample standard deviation[4] of the observed values in the oracle trajectory for the program element; we consider the distribution a reasonable approximation of the distribution of the values that the program element may have. We then sample a random value from this Gaussian distribution; if the sampled value (e.g., a floating point number) is outside the range of the original type (e.g., an integer), we round it to the nearest value in the range. 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 program element and subsequently sample a random string of that length. To avoid sampling the same value repeatedly or sampling the value that is the same as the original value (i.e., no mutation), we keep track of the sampled values and reject any sampled value that has already been used or is the same as the original value.

For most program elements, we gather up to a fixed limit, N_{mpn}, of samples per program element. However, for boolean program elements and program elements with small predetermined value ranges, we sample each value only once, limiting the number of samples collected for these program elements. We leave more refined data mutation and generation strategies to future work and note that techniques for test generation can likely be used to handle more complex, structured data types (Feldt and Poulding, 2013). Since each program element may have a different number of mutated values, we normalize the probabilities in Def. 5.2 and 5.3 by multiplying by the reciprocal of the number of sampled mutated values. For example, each observation for an integer-type program element mutation with ten mutation values has a weight of 0.1.

6. Structure discovery

A few methods already exist to discover causal structure given data, including Bayesian network learning (Singh and Valtorta, 1995; Scutari et al., 2019) and other recent causal discovery algorithms (Rantanen et al., 2018, 2020b,a). However, current algorithms are either unable to handle observations from interventions in the environment or do not scale to the size of the program dependency space, which often consists of more than hundreds of program elements. In this section, we introduce novel methods to discover the causal structure of a program. We first describe the notion of the causal structure in terms of program dependence analysis and the requirements for the discovery of the causal structure. We then introduce two concrete methods to discover the causal structure of a program designed to meet these requirements with respect to different aspects of the qualification of causal relations: the probabilistic aspect and the discrete/deterministic aspect.

6.1. Causal Structure of a Program

The conditional probability defined in Def. 5.3 expresses the *association* between behavior changes to program elements: these are behavior changes that

[4] If there is only one unique observed value, we use 1 for the sample standard deviation.

Journal Pre-proof

are simply observed together. To elevate the association to a *causation*, we need the concept of one's behavior change *preceding* another. It is also necessary to distinguish between *direct* predecessors, nodes whose behavior change affects the target node without involving any intermediary nodes, and *indirect* predecessors, nodes whose behavior change reaches the target node 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 behavior change at S_i will precede the behavior change at S_j. In theory, a perfectly accurate PDG can serve as the causal structure. However, in practice, the required static dependence analysis used to produce a PDG yields many false-positives. Instead, we use ideas from the causal inference field and dynamically approximate the causal structure from the set of observations.

Let us first formally define the predecessors of a node, which we call the *intervention ancestors (IA)*. Given Node S_j in the program, the intervention ancestors of S_j, IA_j, are a set of nodes whose mutation changes the behavior of S_j for at least one input.

Definition 6.1 (Intervention ancestor (IA)). *For a Program $\mathcal{P}$ and a set of nodes S from $\mathcal{P}$, the set of nodes $IA_j \subseteq S \setminus \{S_j\}$ is the set of* intervention ancestors *of node S_j if mutating any node in IA_j changes the behavior of S_j for at least one input. In other words, $S_k \in IA_j$ if and only if there exists a mutation of S_k and an input that causes $S_j = 1$.*

```
A = 42
B = A % 2
C = B + 24
D = A + C
```

S_j	IA_j	MP_j
A	$\varnothing$	$\varnothing$
B	{A}	{A}
C	{A, B}	{B}
D	{A, B, C}	{A, C}

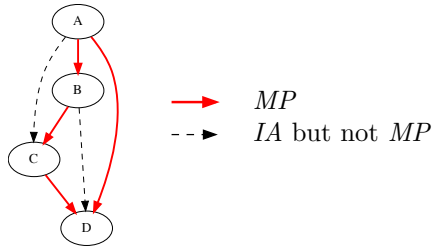


Figure 6: Example of Intervention Ancestors (IA) and Markovian Parents (MP)

A behavior change in one of a node's intervention ancestors will, by definition, precede the behavior change in the node. However, the precedence relationship may be direct (i.e., not pass through any other predecessor) or indirect (i.e., pass through a predecessor). Based on the theory of causal inference, we define the *Markovian parents (MPs)* of a node as the *minimal* set of direct predecessors among the intervention ancestors. Notice that there can be multiple subsets of intervention ancestors that meet the definition of Markovian parents. Figure 6 illustrates intervention ancestors and Markovian parents using a simple program. In the graph, the directed edge $X \to Y$ (both solid-red and dashed-black edges) represents that X is an intervention ancestor of Y; thus, a value change at X leads to a value change at Y. Among the intervention ancestors,

Journal Pre-proof

solid-red edges represent the Markovian parents. For example, changing `A` in Figure 6 may cause a change to the value at `C`, but only subsequently after the value at `B` changes; if the value at `B` does not change, then the value at `C` does not change even though the value at `A` changes. Similarly, `B` is not a Markovian parent of `D`. The Markovian parent-child relation provides the *causal structure* of the program. For instance, the graph with only solid edges in Figure 6 is the causal structure of the sample program.

The definition of Markovian parents leads to the following requirements for the parent-child relations in the causal structure of a program:

- **Requirement 1** → the parent candidates include all nodes whose mutation can change the child's behavior,
- **Requirement 2** → if a child's behavior has changed but the child was not mutated, one of its parents must also have changed behavior, and
- **Requirement 3** → each parent individually has an effect on the child.

The first requirement captures a core principle of dependence: if none of the mutations to program variable A lead to a change in the behavior of variable B, then B does not depend on A. The second requirement also captures a core principle of dependence: if none of the parents' behaviors are changed, the child's behavior cannot be changed unless it is mutated. The last requirement captures direct dependence. If variable A directly affects variable B, it should individually affect B. In other words, there should be a path of effect from A to B without interpolating other program elements.

```
A = 42
B = A + 2
C = B % 2
D = A + C
```

S_j	IA_j	MP_j
`A`	$\varnothing$	$\varnothing$
`B`	{`A`}	{`A`}
`C`	{`A`, `B`}	{`B`}
`D`	{`A`, `B`, `C`}	{`A`, `C`} or {`B`, `C`}

Figure 7: Example of a potential inconsistency between Markovian Parents and actual dependency in a program

The discovered Markovian parents of a node S_j may not be the same as the actual parents of S_j, i.e., the set of program elements whose value directly affects the value/execution of the program element S_j in terms of the program's semantics, either because of the insufficient observation samples or because more than one Markovian parents set exist. For example, consider a variant of source code in Figure 6 where '`B = A + 2`' and '`C = B % 2`' as Figure 7 shows. Then, whenever the value of `D` changes due to the value change of `A`, the value of `B` also changes. In this case, either {`A`, `C`} or {`B`, `C`} can be the Markovian parents of `D`; if the latter is chosen for the Markovian parents of `D`, then the structure discovery algorithm produces both the false-positive (add a program element that is not an actual parent of the child to the Markovian parents) and the false

Journal Pre-proof

negative (miss the actual parent of the child in the Markovian parents). Also, if every mutation on B has the same parity (even or odd) as the original value of B, then $\mathtt{B} \notin IA_{\mathtt{C}}$, and, thus, the false negative result is produced.

Both the false-positive and the false negative for Markovian parents can spoil causal program dependence analysis. A false negative hinders inferring the correct degree of causation from the association. While causal inference derives the causation from the association by taking away non-causal association appearing through a backdoor path (as described in Section 3.3), missing Markovian parents may lead causal inference to ignore the existing backdoor path and infer a less accurate result. For instance, in the example of Figure 3, if we do not know that drinking a lot last night is a cause of the headache, there is no way that the causal inference determines that there is no causal relation between the headache and sleeping with shoes on. A false-positive can harm the causal inference both in terms of accuracy and efficiency. As we observed in the example of Figure 7, if an additional node is mistakenly included in the Markovian parents of a node, it can cause another actual parent of the node to be missed in the Markovian parents. Therefore, the causal inference would infer that there is no causal effect from the actual parent node to the child node. Even if none of the actual parents of the child node is missed in the Markovian parents, the number of causal relations to be inferred for the child node doubles whenever there is a single false parent, as the cardinality of the Markovian parents set is increased, seriously harming the efficiency of the causal inference.

To avoid Markovian parents having false-positives or false negatives, we tried to provide sufficient observations in the experiment (Section 10) and apply heuristics to choose the Markovian parents that are more likely to be the actual parent of the child as described in the following section.

In the remainder of this section, we introduce two causal discovery methods based on the definition of Markovian parents and the three requirements. The first considers the probabilistic aspect of the Markovian parents; in particular, it utilized conditional independence to find the Markovian parents. In addition, we also introduce a variation of the first method that relaxes the requirement of conditional independence to the requirement of similar probabilities. The second method considers the requirement of the Markovian parents in a deterministic and discrete manner interpreting the causal discovery problem as a *hitting set* problem.

6.2. Probability-based Structure Discovery

6.2.1. CP-method

The first method utilizes the definition of Markovian parents in the probabilistic causal model. This definition in probabilistic notation is as follows:

Definition 6.2 (Markovian parent (MP)[5]). *For a set of nodes S, the Markovian parents of Node $S_j \in S$, $MP_j \subseteq IA_j$, is a minimal set of predecessors of S_j that renders S_j independent of all its other intervention ancestors. In other words, MP_j is any subset of IA_j such that $P(s_j \mid mp_j) = P(s_j \mid ia_j)$ while no other proper subset $T \subsetneq MP_j$ satisfies $P(s_j \mid t) = P(s_j \mid ia_j)$.*

Journal Pre-proof

Therefore, the Markovian parents of a node S_j is a minimal subset of its intervention ancestors that preserves the conditional probability of S_j. By computing conditional probabilities with various combinations of intervention ancestors, we can discover the Markovian parents of a node. We call this method the *conditional probability-based method*, i.e., the *CP*-method.

Algorithm 1: Get Markovian parents from intervention ancestors using the *CP*-method

Input: IA_j: Intervention ancestors of S_j,
$Dist$: Distance map from S_j to other nodes,
O: Observations generated from inputs that cover S_j in the original program
Output: MP_j: Markovian parents of S_j

$MP_j \leftarrow \{\}$
$Cand \leftarrow IA_j$
while $Cand \neq \{\} \wedge Cand \neq MP_j$ **do**
  $Remain \leftarrow Cand \setminus MP_j$
  $S_d \leftarrow \text{argmax}_{S_i \in Remain} Dist(S_i)$ ▷ `Get a single element from` $Remain$ `to examine`
  $S_{other} \leftarrow Cand \setminus \{S_d\}$
  $may\_be\_parent \leftarrow False$
  if $S_{other} = \{\}$ **then**
    if $P_O(S_j = 1 \mid S_d = 0) \neq P_O(S_j = 1 \mid S_d = 1)$ **then**
      $may\_be\_parent \leftarrow True$
  else
    $Val_{S_{other}} \leftarrow \{v \mid v \in O|_{S_{other}}\}$ ▷ `Get a unique observation set of` S_{other} `in` O
    foreach $s_{other} \in Val_{S_{other}}$ **do**
      if $P_O(S_j = 1 \mid S_d = 0, s_{other}) \neq P_O(S_j = 1 \mid S_d = 1, s_{other})$ **then**
        $may\_be\_parent \leftarrow True$
        break
  if may_be_parent **then**
    MP_j.ADD(S_d)
  else
    $Cand$.REMOVE(S_d)
    $MP_j \leftarrow \{\}$
return MP_j

However, it requires exponential work to compute the conditional probability as defined by Def. 6.2 for every possible combination of intervention ancestors. Thus, in practice, we approximate the Markovian parents by iteratively removing non-Markovian parents from the intervention ancestors. Algorithm 1 shows

Journal Pre-proof

the process of removing non-Markovian parents from IA_j. We choose one Node S_d from IA_j and check whether S_j is independent of S_d, given all other candidate nodes.

If S_d is the only candidate node left (Line 8-10), we check whether S_j is independent of S_d. If the conditional probability of $S_j = 1$ differs depending on whether S_d is changed (1) or unchanged (0), S_d is a Markovian parent of S_j ($may\_be\_parent = True$). If there are other candidate nodes S_{other} (Line 11-16), we check the conditional independence of S_j from S_d for all observations of S_{other} ($Val_{S_{other}}$). If S_j is conditionally independent of S_d for all s_{other}, S_d is not a Markovian parent. Otherwise, S_d *could be* a Markovian parent of S_j (Line 15). This is because S_j can be conditionally independent of S_d if S_{other} changes. Therefore, we re-check all possible Markovian parents every time we find a new non-Markovian parent (Line 19-21). To minimize the re-checking cost, we order the candidate nodes and first choose the node most unlikely to be the Markovian parent (Line 5). Our distance metric *Dist* represents how likely a candidate is to be a Markovian parent of S_j. For a given Node S_j, the intuition is that a node that appears *close* before S_j in the program execution is more likely to be the Markovian parent of S_j. Thus, nodes that appear after S_j are firstly chosen in Line 5 since they are farthest from S_j, followed by nodes that appear earlier in the execution.

6.2.2. Drawbacks of the CP-method

During early experimentation, we uncovered several drawbacks to the *CP*-method, which can lead it to produce a causal structure that is inconsistent with the actual program dependence.

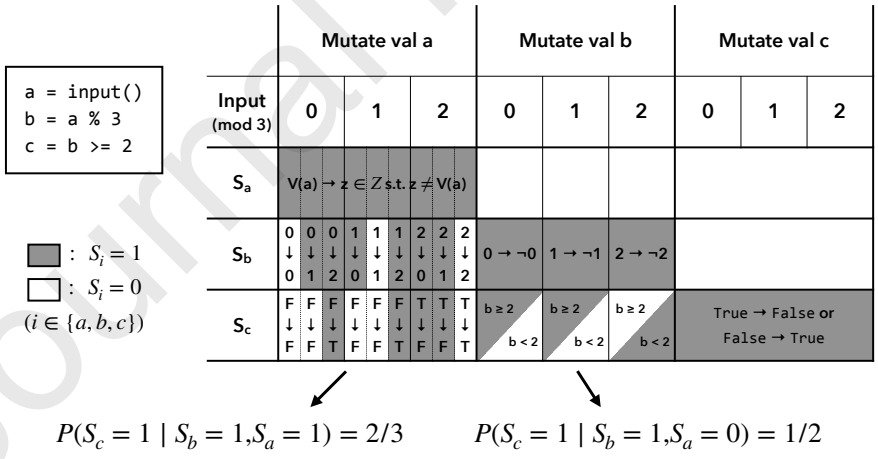


Figure 8: Inconsistency between program dependence and the *CP*-method

- **Inconsistency between program dependence and Markov condition**: the definition of Markovian parents may not be consistent with the

Journal Pre-proof

notion of program dependency in a Δ-execution model. Figure 8 shows an example of this inconsistency. The code on the left shows two data dependencies in the program: `a` → `b` and `b` → `c`. The table on the right side shows the Δ-execution model of the program; the dark color indicates that the value of the variable is changed ($S_i = 1$ ($i \in \{a, b, c\}$)), and the white color indicates the value of the variable is not changed ($S_i = 0$) after the mutation. The text in the cell represents how the value of the variable changes because of the mutation. For example, when mutating `a` (the cell of row S_a under '`Mutate val a`'), `a` changes to an arbitrary integer different from its original value V(`a`), and `b` changes from 0, 1, or 2 to 0, 1, or 2 (the cell of row S_b under '`Mutate val a`').

In cases when the value of `a` is mutated, the value of `b` changes in six-ninths of the cases, and the value of `c` changes in four-ninths of the cases; thus, the conditional probability $P(S_c = 1 \mid S_b = 1, S_a = 1)$ is 2/3. In contrast, when the value of `b` is mutated, it becomes an arbitrary integer different from its original value making the value of `c` changes in half of the cases ($P(S_c = 1 \mid S_b = 1, S_a = 0) = 1/2$). Therefore, regarding the definition of Markovian parents, S_b cannot screen S_c from S_a, leaving `a` → `c` in the causal structure. The two conditional probabilities are different because the value distributions of `b` in the Δ-execution model are different when `a` is mutated (mutated `b` $\in 0, 1, 2$) and `b` is mutated (mutated `b` $\in \mathbb{Z} \setminus \{\text{original } b\}$).

- **Loss of probability precision due to sampling bias**: Regardless of whether the value distributions of the program element are identical, the equality checking of conditional probabilities itself may not lead to finding the correct causal structure. We use the samples from the Δ-execution model during the experiment when computing the conditional probability. Due to the sampling bias, there can be inaccuracies during equality checking.

6.2.3. Relaxed CP-method

The *CP*-method determines that a node S_d has no effect on another node S_j if probabilities of S_j under the conditions of $S_d = 0$ and $S_d = 1$ are the same given the other parent nodes of S_j. However, it is often difficult to expect that two conditional probabilities become the same, due to sampling bias and inconsistency between program dependence and Markovian parents, introducing a large number of false-positive edges to the structure.

One remedy to this problem is to relax the equality checking of conditional probabilities. Our *relaxed CP-method*, *RCP*-method, is an extension of the *CP*-method that takes a constant threshold value and checks whether the difference between the conditional probabilities is less than the threshold. If the difference is less than the threshold, then the *RCP*-method finds that there is no effect from the parent candidate node to the child node. This relaxation makes it easier to remove a parent candidate and reduces the number of false-positive edges. Yet, it can instead introduce false negative edges to the structure. We

Journal Pre-proof

investigate the effect of relaxation on the structure discovery while varying the threshold value as part of our empirical investigation.

6.3. Structure Discovery using Hitting Sets

6.3.1. HS-method

The second method exploits the requirements of the parent-child relation to discover the causal structure. Instead of relying on the probabilistic notion, the *HS*-method formalizes the structure discovery problem as a collection of *hitting set problems*, a classical question in combinatorics. Given a ground set of elements U and a collection C of subsets of U, the hitting set problem is to find the smallest subset H of U such that H hits (includes an element of) every set found in C. In the formal notation, a hitting set H of $C \subseteq 2^S$ is a minimal subset of S satisfying

$$\forall E \in C, H \cap E \neq \varnothing.$$

In the *HS*-method, finding a set of Markovian parents of a node S_j is equivalent to a single hitting set problem. The description of the hitting set problem corresponds to each of the requirements from Section 6.1:

- **Requirement 1** $\rightarrow$ A ground set of elements $U = IA_j$, intervention ancestors of S_j.
- **Requirement 2** $\rightarrow$ For each observation $o \in O$ whose S_j's behavior has changed without mutating S_j, there is a corresponding subset in the collection C, where the subset elements are the intervention ancestors that changed together with S_j in o; and vice versa.
- **Requirement 3** $\rightarrow$ The Markovian parents of S_j, MP_j, is a hitting set of C.

Algorithm 2 shows the algorithm for the *HS*-method. Similar to the *CP*-method, the *HS*-method first identifies the intervention ancestors IA_j of a node S_j. Then, given observations O from a Δ-execution model, it identifies a subset *SubS* of IA_j whose value also changed along with S_j (Line 7). Unless S_j is the only node that changed, one of the parent nodes of S_j should also be changed and thus be included in *SubS*. Solving a hitting set problem on the collection of *SubS* returns a minimal parents of S_j satisfying the above requirements (Line 11). There may be several subsets of IA_j satisfying the hitting set conditions. To cope with this, the *HS*-method uses the same distance metric as the *CP*-method to choose the most plausible Markovian parents. We add a constraint to the hitting set problem to prefer the Markovian parents whose sum of the weights, which is proportional to the distance metric from S_j, is the smallest.

6.3.2. Advantages of the HS-method

The *HS*-method has the following advantages over the (*un-relaxed*) *CP*-method:

Journal Pre-proof

Algorithm 2: Get parents in the causal structure from intervention ancestors using the *HS*-method

Input: IA_j: Intervention ancestors of S_j,
$Dist$: Distance map from S_j to other nodes,
O: Samples from Δ-execution model
Output: MP_j: Parents of S_j in the causal structure

```
if IA_j = {} then
    return {}
else
    Collection ← {}
    foreach o ∈ O do
        if o[S_d] = 1 then
            SubS ← {S_i | S_i ∈ IA_j ∧ o[S_i] = 1 ∧ S_i ≠ S_j}
            if SubS ≠ {} then
                Collection.add(SubS)
    WeightMap ← {S_i : 1 + Dist(S_i) × 10^−6 | S_i ∈ Dist}
    return HittingSet(Collection, WeightMap)
```

- **More suitable to a Δ-execution model**: the rules of the *HS*-method are deterministic; instead of comparing the conditional probability, the hitting set-based algorithm checks the inclusive relationship between the parent candidates. It is more suitable for a Δ-execution model because it is independent of the detailed difference between value distributions from different mutations. We expect this to lead the *HS*-method to produce a causal structure closer to the program dependence structure. For example, in the case of Figure 8, whenever the values of `a` and `c` change, the value of `b` changes as well; thus, the *HS*-method removes S_a from the parent set of S_c.

- **Less sensitive to sampling bias**: the *CP*-method is excessively sensitive to sampling bias as it requires two conditional probabilities to be identical, which could easily become false when we estimate the probability from the samples. On the other hand, the *HS*-method does not calculate the probability. It only considers whether there is another parent candidate that always changes when one parent candidate and the child change. Thus, it is much less sensitive to the sampling bias than the original *CP*-method.

7. Causal Program Dependence Model

In this section, we apply two metrics to measure the strength of the dependency in CPDA. The first, *average causal effect* (Holland, 1988), measures the total effect of each node's change that causes a change in another node. The

Journal Pre-proof

second, *natural direct effect* (Pearl, 2001), measures the effect of one node on another excluding all indirect effects through other nodes. Finally, we build the causal program dependence model (CPDM), a graphical representation of the program dependence based on the output of CPDA.

7.1. Average Causal Effect (ACE)

Given a causal structure, causal inference can measure the degree of causation with the causal effect $P(y|do(x))$, a probability of Y having y caused by setting X to x. Yet, to quantify the dependence, we need to measure any difference in the behavior of the affected program element due to the change of the affecting program element. For instance, if the behavior of program element S_j is always changed no matter how we change the behavior of S_i, i.e., $P(S_j = 1 \mid do(S_i = 1)) = P(S_j = 1 \mid do(S_i = 0)) = 1$, then S_i has no effect on S_j.

Average causal effect (Holland, 1988) measures the total effect of the change of one random variable X on another random variable Y by subtracting the causal effect on Y when X is set to different values.

$$ACE(X \to Y) = P(y \mid do(x_1)) - P(y \mid do(x_0)).$$

In CPDA, we apply the average causal effect to measure the total effect of the causal dependency, i.e., the total effect of the behavior change of S_i on the behavior change of S_j. The average causal effect in CPDA (ACE) is defined as follows:

Definition 7.1 (Average Causal Effect in CPDA (ACE)). *Given a set of observations O and two nodes S_i and S_j, the average causal effect in CPDA (ACE) from S_i to S_j, $ACE_O(S_i, S_j)$, is defined as follows:*

$$ACE_O(S_i, S_j) = P_O(S_j = 1 \mid do(S_i = 1)) - P_O(S_j = 1 \mid do(S_i = 0)).$$

7.2. Natural Direct Effect (NDE)

Natural direct effect (Pearl, 2001) quantifies the portion of the effect that is not mediated by any other 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.

Definition 7.2 (Natural Direct Effect). *The natural direct effect, denoted as $nde_{X:x\to x'}(Y)$ is the expected existence of a change in Y induced by changing X from x to x' while keeping all mediating factors constant at whatever value they would have had under $do(x)$. $nde_{X:x\to x'}(Y)$ is calculated as follows:*

$$\sum_z \left[E\left(Y \mid do(x', z)\right) - E\left(Y \mid do(x, z)\right)\right] P\left(z \mid do(x)\right),$$

where E is the expectation operator, and Z represents all parents of Y excluding X.

Journal Pre-proof

Similar to ACE, we apply the natural direct effect to quantify only the direct effect of the behavior change of S_i on the behavior change of S_j. The natural direct effect in CPDA (NDE) is defined as follows:

Definition 7.3 (Natural Direct Effect in CPDA (NDE)). *The natural direct effect in CPDA (NDE) from S_i to S_j, denoted as $NDE_{\mathcal{I}}(S_i, S_j)$, is the average of the natural direct effect in Definition 7.2 from S_i to S_j over all inputs $\mathcal{I}$:*

$$NDE_{\mathcal{I}}(S_i, S_j) = \frac{1}{|\mathcal{I}|} \sum_{t \in \mathcal{I}} nde_{O_t, S_i:0\to1}(S_j)\,,$$

where $nde_{O_t,S_i:0\to1}(S_j)$ denotes the natural direct effect in Definition 7.2 given O_t, the observations from input t, for computing the probability.

7.3. Causal Program Dependence Model

Given the definition of the NDE, CPDM is a weighted dependence graph for the program. Its structure is the causal structure where each edge weight denotes the NDE between the nodes. CPDM is thus a novel graph representation that explains program dependence in a continuous, gradual way.

8. Experimental Setup

This section presents the empirical evaluation we designed to investigate how CPDA and CPDM could identify and quantify the program dependence and the potential of the quantified dependency to aid various downstream tasks. Subsequently, it introduces the subject programs we consider, some baselines, and the implementation environment.

8.1. Research Questions

Our research questions can be organized into two different sets. The first set investigates the structure discovery methods. We evaluate the performance of the *CP*-method and the *HS*-method in terms of the accuracy of the causal structure compared to the ground truth PDG and the efficiency of discovering the structure.

RQ1-1. Causal Structure and PDG: *How accurate are the causal structures discovered by the CP-method and the HS-method compared to the ground truth PDG?* We examine the accuracy of the causal structure by counting the number of false positive edges (the structure contains an edge that is not in the ground truth PDG) and the number of false-negative edges (the structure does not contain an edge that is in the ground truth PDG). We also qualitatively analyze the root cause of such discrepancies.

RQ1-2. Efficiency: *How efficient are the structure discovery algorithms?* It is worth noticing that the dependence analysis often serves as a pre-processing step for downstream analysis. Therefore, the efficiency of the dependency

Journal Pre-proof

analysis is a critical factor in its usefulness. We compare the efficiency of the *CP*-method and the *HS*-method using multiple subject programs. We report the average time spent running each method ten times.

RQ1-3. *RCP*-**method:** *How does the relaxation affect the performance of the CP-method?* To overcome the drawbacks of the *CP*-method we extend the *CP*-method by introducing a threshold to relax the criterion of dependence (*RCP*-method). We compare the performance of the *RCP*-method using various thresholds.

The second set of research questions investigates the characteristics and potential benefits of CPDM's quantified program dependence. For evaluation, we present three scenarios to show the utility of CPDM. The first two have a straightforward comprehension focus. The *Quantified Dependence* scenario uses CPDM to illustrate program semantics via clustering the strongly connected (i.e., strongly dependent) program elements, while the *Execution Awareness* scenario uses the CPDM to identify execution scenarios via observational subsetting. Finally, our *Debugging* scenario considers how CPDM can assist in debugging based on quantified dependence.

RQ2-1. Quantified Dependence: The strength of the dependency relations in a program can vary considerably depending on a program's semantics. A key characteristic of CPDA is its causal inference-based estimates of dependence strength. The first scenario asks the question, *does the CPDM capture dependence strength sufficiently to assist in understanding a program's semantics?* For evaluation, we undertake CPDA for several subject programs and examine the CPDM from two different viewpoints. First, we consider the whole CPDM of a program, clustering the nodes using dependence strength and checking how this can express the functional aspects of the program. Second, we take a closer look at each program element and how the relative degree of dependence to and from its neighbors helps us understand its semantics. In addition, we investigate whether CPDM can accurately identify the non-existence of dependences, which are often produced as false-positive dependences by the static analysis.

RQ2-2. Execution Awareness: A program may have several execution scenarios with different functionalities and dependences. CPDA can estimate the dependence on a specific execution, or for some subset of executions, by simply choosing the corresponding (subset of) observations. In this scenario, we ask *how does the estimated CPDM from different execution scenarios aid in program comprehension?*

RQ2-3. Debugging: In the final scenario, we consider *how quantified dependence estimated by CPDA can be employed when debugging.*

8.2. Subject Programs

To explore aspects of CPDA we make use of three subject programs with easily understood semantics: Triangle (tri), a classic subject from software test-

Journal Pre-proof

ing, word count (wc), a widely studied program in the program dependence literature (Gallagher and Lyle, 1991; Lee et al., 2021), and Bill&Ted (B&T), a program with more complicated control flow structure than tri and wc. In addition we consider tcas, a more substantial program designed to determine if two airplanes are headed for a collision. We manually construct each program's PDG regarding the program elements (defined in Section 5.2) and use it as the ground truth. Finally, in order to capture various execution scenarios in the experiments, we consider a range of test suites. We employ all the tests to estimate the CPDM in the quantified dependence scenario (**RQ2-1**), while we consider several subsets in the execution awareness scenario (**RQ2-2**). We describe these four subjects in greater detail and then briefly describe a few programs and techniques used for specific purposes in the empirical exploration.

- **Triangle (tri)**: Given three natural numbers corresponding to the length of each side of a triangle, Triangle determines whether the sides form a triangle or not, and if so if the triangle is equilateral, isosceles, or scalene. The universal test input space considers the values 1-5 for each side; it consists of $5^3 = 125$ test cases in total. We choose a maximum length of 5 since a maximum of 3 constructs no scalene triangles, and a maximum of 4 yields only a single scalene triangle (of sides 2, 3, and 4). Along with the ***total*** test suite, we use two additional test suites for tri:

  valid : Tests where the sides satisfy the triangle inequality (65 of the 125 test cases).

  ordered : Tests where the sides are ordered in non-decreasing order (35 of the 125 test cases).

  While restricting the input to only valid sides significantly changes the distribution of triangle types, ordering of the sides causes each side to play an asymmetric role; for example, when two sides are equal, the middle side length will always be one of the equal sides.

- **Word count (wc)**: The word count program counts the number of characters, lines, and words found in its input text. We manually generate four test suites that combined with an empty input (Test 0) provide branch adequate coverage:

  ***onechar*:** Test 1 contains a single letter.

  ***oneword*:** Tests 2-3 contain a single word of multiple letters.

  ***oneline*:** Tests 4-5 contain a single line with multiple words.

  ***multiline*:** Tests 6-7 contain multiple lines of multiple words.

- **Bill&Ted (B&T)**: Bill&Ted is a program designed to compute the parking fee for several classes of vehicles (e.g., cars, trucks, etc.) staying for various lengths of time. We manually generated a path-coverage adequate test suite consisting of 114 tests.

Journal Pre-proof

- **TCAS Version 1 (tcas-1)**: To evaluate the debugging task, we employ a buggy version of the program, TCAS Version 1 (tcas-1), from the Siemens suite (Do et al., 2005). As is common in such studies, we assume a single failing test in a minimal statement-coverage adequate test suite.

To investigate CPDA's ability to correctly establish the non-existence of dependences, we employ two additional programs, mbe and mug, from work by Binkley et al. (2015). These two demonstrate the limitation of static analysis, which commonly produces false-positive dependences for these examples. Binkley et al. (2014) show that observation-based analysis (observation-based slicing (ORBS)) can successfully identify the true negative dependences. We use these examples to investigate if, like ORBS, CPDA can correctly avoid these potential false-positive dependences.

While we choose vetted programs for evaluating the CPDA's causal dependency, the size and complexity of the programs are not varied to investigate the accuracy and scalability of our structure discovery algorithms. To address this, we study artificial DAGs with a given number of nodes and random edges for the structure discovery evaluation in addition to the above subject programs. Given a graph, a sensitivity (the probability of a child change given one of its parent changes), and a mutation count (the number of times a node is mutated), we generate samples of which nodes change together if one changes one of the nodes in the graph. We consider three artificial graphs (each of n, e (r) denotes the number of nodes, edges e, and the edge ratio r such that $e = (n) \times (n-1) \times r$):

- G_1: 5, 4 (0.2)
- G_2: 50, 98 (0.04)
- G_3: 500, 998 (0.002)

8.3. Baselines

We explore the characteristics of CPDA and CPDM against the use of static dependency found in a PDG (Ferrante et al., 1987). Using a debugging task, we also investigate how the quantified dependence from CPDA differs from that of the PPDG (Baah et al., 2010) and BNPDG (Yu et al., 2016) mentioned in the motivating example of Section 2. To do so, we apply their work on dynamic dependence analysis to the fault localization problem. The main hypothesis of PPDG's fault localization is that the reaching definition with the smallest conditional probability in PPDG learned from passing executions that occur in the failing test execution is deemed the most suspicious and, thus, the most likely to have a fault. The BNPDG, on the other hand, assumes that a node is more likely to be faulty if the conditional probability of the node given the state of an erroneous output is higher. We compare their ability to detect the fault compared with that of CPDM as part of **RQ2-3**.

Journal Pre-proof

8.4. Implementation, Configuration, and Environment

The theory of causal inference assumes that the structure of the model is a DAG and thus acyclic. To apply CPDA on a program with loops, we unroll the loops of the program. After applying CPDA to the unrolled program, we merge nodes that represent instances of the same program element. Merging allows a self-loop, which represents a program element dependent on its previous value. After the merge, the ACE (or NDE) between Node A and Node B is the maximum of the ACE (or NDE) between any instance of Node A and any instance of Node B. The base idea of using the maximum is that if an instance of Node A always affects an instance of Node B in the unrolled program, then we can say that Node A always affects Node B in the original program.

To select nodes for analysis and to insert logging functions in the original program we use srcML (Collard et al., 2013), an open-source tool that parses the source code into an XML format. After performing a preliminary experiment with CPDA to choose a sufficient number of mutation samples (N_{mpn}) that gives steady empirical results, we use 100 mutations ($N_{mpn} = 100$) for two smaller programs, tri and wc, and 20 mutations ($N_{mpn} = 20$) for B&T and tcas for the rest of the experiments. We choose various thresholds including 0, powers of 0.1, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1} (0.1), and 0.2, 0.3, 0.5 to evaluate the *RCP*-method. Then, we further choose thresholds 0.0002, 0.0004, 0. 0006, 0.0008, 0.002, 0.004, 0.006, and 0.008 for a more precise experiment as we observe the accuracy of the structure varies largely around thresholds of 0.001.

All experiments are performed using Ubuntu 18.04 on an Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz with 32GB of RAM. Computing the conditional independence in the *CP*-method can be run in parallel. Our experiment uses Nvidia Titan X for parallelized computation. All the experimental data are available in `https://to.be.released`.

9. Results

This section presents the empirical evaluation of our causal program dependence analysis.

9.1. RQ1. Structure Discovery

9.1.1. RQ 1-1: Causal Structure and the PDG

Figures 9, A.22, A.23, and A.24 show the causal structure discovered by the *HS*-method and the *CP*-method for the four benchmark programs tri, B&T, wc, and tcas.[6] While both causal structures for tri are the same and consistent with the ground-truth PDG, for the other three subject programs the causal structures for the *CP*-method have significantly more edges than that for the *HS*-method.

[6] Figure A.22, A.23, and A.24 are in Appendix Appendix A due to their size.

Journal Pre-proof

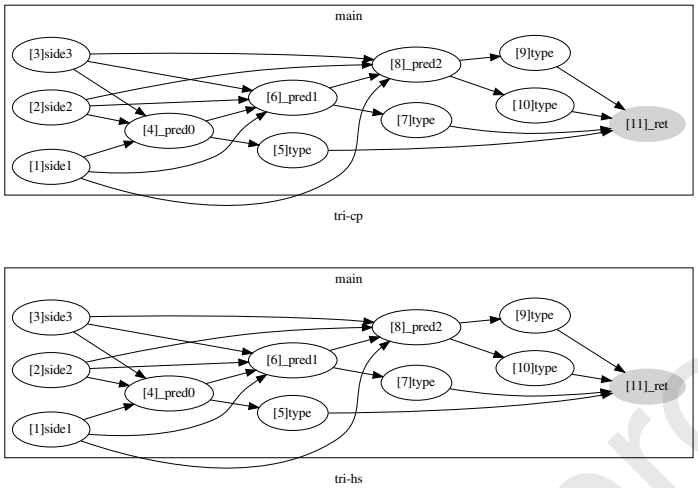


Figure 9: Causal structure of tri (left: the *CP*-method; right: the *HS*-method)

TP/FP/FN	tri	B&T	wc	tcas
CP-method	19/0/0	108/93/4	28/7/1	92/55/10
HS-method	19/0/0	106/2/6	27/0/2	79/7/23

Table 1: Accuracy (TP: true-positive / FP: false-positive / FN: false-negative) of causal structure with respect to the PDG

Table 1 shows the accuracy of the two causal structures compared to the ground truth PDG. Overall the *HS*-method produces significantly fewer false-positive edges (non-dependency edge in the causal structure) compared to the *CP*-method. Furthermore the *HS*-method produces no false-positive edges for tri and wc and only two and seven, respectively, for B&T and tcas. On the other hand, while the *CP*-method produces no false-positive edges for tri, it produces seven for wc and more than fifty for B&T and tcas. Thus the *HS*-method produces significantly fewer false-positive edges than the *CP*-method, however, on the flip side it produces a few more false-negative edges (no dependency edge in the causal structure) than the *CP*-method.

```
1 if (p) {
2   x = f();
3   y = g(x);
4 }
```

```
1 a = ...;
2 p = f(a);
3 q = g(p);
4 if (q)
5   r = h(a);
```

Figure 10: Example of a false-negative edge (left) and a false-positive edge (right) in the causal structure

Journal Pre-proof

The left side of Figure 10 shows a typical example of a false-negative edge in the causal structure. In the figure, the variable `y` is control dependent on the variable `p`. However, whenever `p` is mutated, the value of `x` also changes. Therefore, the *HS*-method discards the edge from `p` to `y`. This suggests that separating edges whose source is a control node from other edges might bring value as it should separate control and data dependence edges. We will discuss this in the future work section.

A large number of the false-positive edges in the *CP*-method's causal structure are due to the inconsistency between program dependence and the Markov condition we discussed in Section 6.2.2. The right side of Figure 10 shows a typical example of a false positive edge we found in the *HS*-method's causal structure of tcas, which is also identical to what is described in Section 6.1. The true Markovian parents of `r` at Line 5 are `q` and `a` from Lines 1 and 2, while `p` in Line 3 is also one of the parent candidates. In the observation, a change of `a`'s behavior always changes `p`'s behavior. At the same time, there is an observation where `a` and `p`'s behavior changed, but `q` is unchanged from True, therefore leading `r`'s behavior to change. Then, due to the distance-based heuristics, the structure discovery algorithm prefers `p` as a Markovian parent above `a`, creating the false-positive edge.

Summary of RQ1-1: The *HS*-method produces significantly fewer edges compared to the *CP*-method. Compared to the ground truth PDG of the benchmark program, the *HS*-method produces significantly fewer false-positive edges than the *CP*-method. However, it produced slightly more false-negative edges.

9.1.2. RQ 1-2: Efficiency

Time (s)	tri	B&T	wc	tcas	G_1	G_2	G_3
CP-method (w/o GPU)	0.7	13,281.3	44.9	40.7	0.0033	1.3	1600
CP-method (w/ GPU)	1.8	1,028.5	48.8	34.7	0.021	0.68	19
HS-method	0.6	23.5	8.6	1.5	0.00089	0.020	0.39

Table 2: Time spent (seconds) during Algorithm 1 and Algorithm 2 for the four benchmark programs and three artificial graphs

Next, we compare the efficiency of the *CP*-method and the *HS*-method in terms of the time spent for the four benchmark programs, tri, B&T, wc, and tcas, and the artificially generated graphs. The second to the fifth columns of Table 2 compare the time spent calculating the causal structure of the four benchmark programs using the *CP*-method and the *HS*-method. We only compare the wall-clock time of Algorithm 1 and Algorithm 2 without the time getting the intervention parents as both methods share it. The result shows that the *HS*-method is faster than the *CP*-method, both with or without using GPU. The difference is the largest for the biggest program, B&T (227 lines), where the *HS*-method is 44 times faster than the *CP*-method with GPU. The difference is moderate for tcas (182 lines) and wc (54 lines) while there is almost no difference

Journal Pre-proof

for tri (20 lines). The sixth to the last columns of Table 2 compare the time spent calculating the causal structure of the three artificial graphs. For this experiment, we assume that the graph is fully sensitive (the child always changes when one of its parents changes) and generate samples by mutating each node once. The result again shows that the *HS*-method is faster than the *CP*-method. Similar to the previous result, the difference becomes larger as the graph size increases. We also can see that the GPU is useless for reducing the time cost when the program/graph size is small.

	$r = 0.01$	0.02	0.04	0.08	0.16
CP-method (w/o GPU)	0.007	0.12	1.3	2.0	2.2
CP-method (w/ GPU)	0.021	0.17	0.68	0.96	1.0
HS-method	0.0093	0.013	0.020	0.029	0.028

Table 3: Time spent during Algorithm 1 and Algorithm 2 for different number of edges

To take a closer look at the effect of the number of edges on the efficiency of the structure discovery algorithms, we run the same experiment using G_2 with a different edge density. Table 3 shows the time spent on graphs with different edge densities. The result again shows that the *HS*-method is faster than the *CP*-method. It also finds that the time cost of the *HS*-method stabilizes by $r = 0.16$, while the cost of the *CP*-method continues to increase.

Finally, we compare the effect of the number of samples. For this experiment, we consider the sensitivity of 0.75 for G_2 for the following reason: the edge ratio of 0.04 in G_2 assigns two incoming edges for each node on average; the sensitivity of 0.75 limits the chance of the parents' effect not propagating to the child by around 5% ($\approx 0.25^2$) when two or more parents change. Then, we mutate each node 1, 5, 10, and 20 times to generate a different number of samples. Along with the time spent, we also present the number of true-positive, false-positive, and false-negative edges compared to the original graph. For comparison, we also include the case of the fully sensitive graph with a single mutation per node.

	Sen=1, Mut=1		Sen=0.75, Mut=1		Sen=0.75, Mut=5		Sen=0.75, Mut=10		Sen=0.75, Mut=20	
	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc	T(s)	Acc
CP(w/o GPU)	1.3	62/0/37	0.93	65/116/34	8.5	94/357/5	20	95/477/4	42	97/585/2
CP(w/ GPU)	0.68	62/0/37	0.64	66/119/33	1.6	92/363/7	6.4	95/482/4	15	97/585/2
HS 0.020	62/0/37	0.016	56/24/43	0.029	89/4/10	0.046	91/4/8	0.073	95/3/4	

Table 4: Time spent (T) and accuracy (Acc: TP/FP/FN) for Algorithm 1 and Algorithm 2 for different number of samples. "Sen" represents the sensitivity and "Mut" represents the number of mutations per node.

Table 4 shows the time spent (T) and accuracy (Acc: TP/FP/FN) for Algorithm 1 and Algorithm 2 for a different number of samples. The data find that the *HS*-method outperforms the *CP*-method in terms of efficiency. Notice that the *HS*-method is also more scalable to the sample increase than the *CP*-method; the *HS*-method's ratio of time spent as the number of samples

Journal Pre-proof

increases is smaller than that of the *CP*-method. The difference in the accuracy between the *CP*-method with and without GPU is due to a CPU versus GPU floating-point difference.

According to Table 4, both the *HS*-method and the *CP*-method add more edges to their causal structure as the size of the sample increases (TP + FP). The number of true-positive edges is marginally larger in the *CP*-method than that in the *HS*-method. However, the number of false-positive edges is significantly larger in the *CP*-method than that in the *HS*-method. The number of false-negative edges is marginally larger in the *HS*-method than that in the *CP*-method. An interesting point to notice is the change in the number of true-positive, false-positive, and false-negative edges. While the number of false-positive edges and the number of false-negative edges decreases in the *HS*-method as the number of samples increases, the number of true-positive edges increases dramatically in the *CP*-method. The difference in the number of false-negative edges between the two methods decreases as the number of samples increases.

Summary of RQ1-2: The *HS*-method is much more efficient than the *CP*-method. As the number of samples increases, the number of false-positive edges and the false-negative edges decreases for the *HS*-method, while the number of false-positive edges increases for the *CP*-method.

9.1.3. RQ 1-3 RCP-method

To evaluate the *RCP*-method, we choose the B&T benchmark program since the *CP*-method introduces the largest number of false-positive edges to the causal structure. We also again use artificial graph G_2 with a 0.75 sensitivity and 20 mutations per node, which is where the *CP*-method performed poorly regarding the number of false-positive edges.

Threshold		0	1e-6	1e-5	0.0001	0.0002	0.0004	0.0006	0.0008	0.001	0.002	0.004	0.006	0.008	0.01	0.1	0.2	0.3	0.5
B&T	TP	108	108	107	105	105	105	105	105	**105**	**103**	96	94	90	81	26	0	0	0
	FP	93	63	43	22	14	13	10	10	**8**	**6**	3	2	2	2	0	0	0	0
	FN	4	4	5	7	7	7	7	7	**7**	**9**	16	18	22	31	86	112	112	112
G_2	TP	97	97	94	91	91	91	**88**	84	**83**	76	64	63	55	48	2	0	0	0
	FP	585	539	300	95	58	33	**18**	16	**13**	12	15	10	11	14	3	0	0	0
	FN	2	2	5	8	8	8	**11**	15	**16**	23	35	36	44	51	97	99	99	99

Table 5: The number of true-positive, false-positive, and false-negative edges for the *RCP*-method for different thresholds. "B&T" and "G_2." represents the B&T benchmark program and artificial graph, respectively. The bold text shows the case when the sum of the numbers of false-positive edges and false-negative edges is the smallest.

Table 5 shows the number of true-positive, false-positive, and false-negative edges found in the causal structure of the *RCP*-method using different thresholds. In general, the number of true-positive edges and the number of false-positive edges decreases while the number of false-negative edges increases as the threshold increases for both subjects, which naturally implies that a larger threshold leads more parent candidates to be discarded. The number of true-positive edges increases when the threshold is from 0.002 to 0.004 and from 0.006 to 0.008. We expect this to happen since the order of parent candidate

Journal Pre-proof

removal affects the final Markovian parent set; removing one parent candidate by increasing the threshold may cause the inability to remove a larger number of parent candidates.

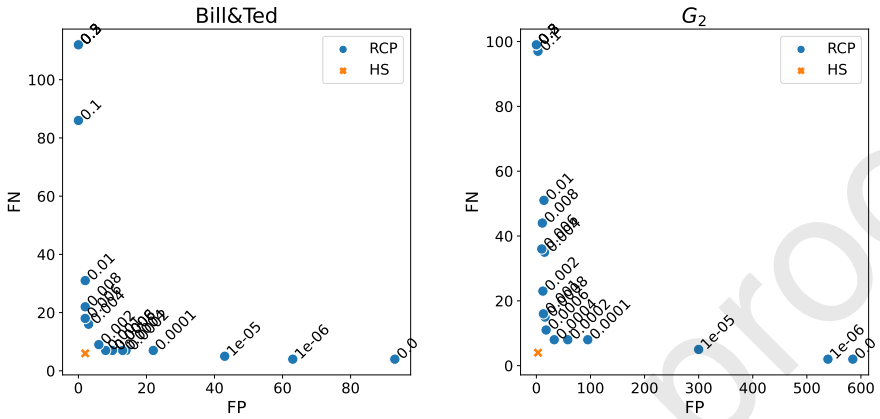


Figure 11: The number of false-positive (FP), and false-negative (FN) edges found in the causal structure of the *RCP*-method using different thresholds and the *HS*-method.

The result shows that the sum of the number of false-positive edges and false-negative edges is smallest when the threshold is 0.001, 0.002 for B&T, and 0.0006, 0.001 for G_2. The number of true-positive edges only loses three (nine) when the threshold is 0.001 (0.006) for B&T (G_2) compared to when there is no threshold. Figures 11 and A.25 present the result in Table 5 as well as the result of the *HS*-method from Table 1 and Table 4.[7] It shows that the *RCP*-method never produces a smaller sum of the number of false-positive and false-negative edges than the *HS*-method. Nevertheless, the result of the *RCP*-method illustrates its potential to control the trade-off between the false-positive and the false-negative in the structure discovery.

Summary of RQ1-3: The result of the *RCP*-method illustrates its potential to control the trade-off between the false-positive and the false-negative in the structure discovery. Yet none of the *RCP*-method of thresholds we investigated produces a smaller sum of the number of false-positive and false-negative edges than the one from the *HS*-method. Based on the result of **RQ1**, we use the *HS*-method in addressing RQ2.

Journal Pre-proof

```
1  def main() {
2     <1>characters, <2>lines, <3>words, <4>inword =
        0, 0, 0, 0
3     while (<5>_pred1 = (scanf("%c", <6>&c) == 1))
        {
4        <7>characters = characters + 1
5        if (<8>_pred2 = (c == '\n'))
6           <9>lines = lines + 1
7        if (<10>_pred3 = isLetter(c)) {
8           if (<11>_pred4 = (inword == 0))
9              <12>words = words + 1
10          <13>inword = 1
11       }
12       else
13          <14>inword = 0
14    }
15 }
16 def isLetter(<15>c) {
17    if (<16>_pred5 = ((c >= 'A' && c <= 'Z') || (
       c >= 'a' && c <= 'z')))
18       <17>_ret = True
19    else
20       <18>_ret = False
21    return _ret
22 }
```

Figure 12: Pseudo-code of the wc subject program with node numbers shown between angular brackets, ⟨n⟩ and predicates explicitly pulled out into assignment statements.

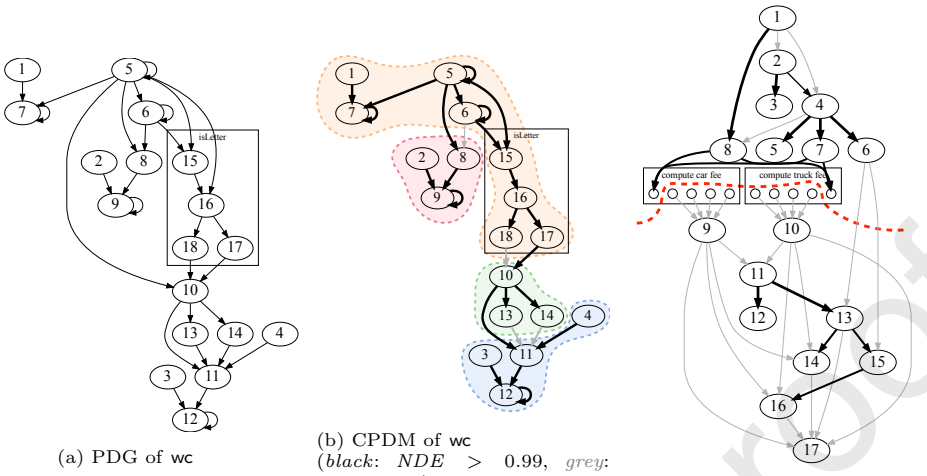
9.2. RQ2. Quantified Dependency

9.2.1. RQ 2-1: Program Semantics Clustering

Figure 12 shows the pseudo-code of wc where node indexes are annotated with angular brackets. Figures 13a and 13b show the resulting PDG and CPDM. The first thing to notice is how similar the two graphs are. This illustrates that causal inference is able to distinguish many of the same dependences as found in a PDG. It does this without the need for the formal semantics of the programming language. Furthermore, as illustrated below, the CPDM *omits* certain unwanted edges present in the PDG. In the CPDM the thickness of the edges reflects the degree of NDE. Clustering program elements based on how strongly they depend on each other reveals patterns that relate to features in the source code. This can be observed in Figure 13b, in which nodes are grouped together based on the average degree of dependence within subgroups. The clusters, shown using dashed lines, highlight nodes with strong connections that correspond to features of the program. For example, nodes ⟨1⟩, ⟨5-7⟩, and ⟨15-18⟩ count the input character and check if it is alphabetic. Similarly, nodes ⟨2⟩, ⟨8⟩, and ⟨9⟩ are involved with the line count, while ⟨10⟩, ⟨13⟩, and ⟨14⟩ capture

[7] Figure A.25 is in Section Appendix A due to the size of the figure.

Journal Pre-proof



(a) PDG of wc

(b) CPDM of wc (*black*: $NDE > 0.99$, *grey*: $NDE > 0.0$)

(c) CPDM of B&T (*black*: $NDE > 0.7$, *grey*: $NDE > 0.2$)

Figure 13: (a) wc's PDG, (b) CPDM and (c) B&T's CPDM where thicker edges reflect larger NDE. The dashed lines represent the cluster of nodes based on the strength of the NDE.

the in-word logic. Finally nodes ⟨3⟩, ⟨4⟩, ⟨11⟩ and ⟨12⟩ count the number of words.

In comparison, the nodes weakly connected (grey edges in Figure 13b) often reflect features that are only occasionally executed by the test suite. For example, the word counting feature is only executed when there is at least one non-alphabet character in the input. Related (weaker) dependences (denoted $\{\langle \text{from} \rangle\} \rightarrow \{\langle \text{to} \rangle\}$) include $\{\langle 18 \rangle\} \rightarrow \{\langle 10 \rangle\}$, $\{\langle 13 \rangle\} \rightarrow \{\langle 11 \rangle\}$, and $\{\langle 14 \rangle\} \rightarrow \{\langle 11 \rangle\}$. Even a feature that is executed in every execution may have a small NDE. For example, the character read at node ⟨6⟩ affects the predicate at node ⟨8⟩, which in turn checks whether the character is a newline or not. Because only newline characters matter the causal relationship is not strong.

Such examples show that the quantified dependences with NDE in the CPDM can capture the aspects of program semantics more concisely than the traditional indistinguishable dependences in the PDG. In contrast, despite being a small 40-line program, wc's PDG (Figure 13a) in which all edges have equal weight makes it challenging to identify computationally related portions of the code.

As a second case study, Figures 14 and 13c show the pseudo-code and the partial CPDM for the main function of the B&T example. The CPDM includes the invocation of two functions (the two rectangles) that compute the fee for cars and trucks: it reveals two clusters of strong dependence that differentiate the functional aspect of the code before and after the call to one of the fee calculation functions. The former cluster captures the preparation ahead of the fee calculation: identifying the type of the vehicle and deciding the charging rule. The latter cluster shows the post-processing applying a discount or a surcharge.

Journal Pre-proof

```
def main(args) {
   <1>car_type = args[0]
   <2>_pred1 = car_type == "SENIOR_CITIZEN"
   if (_pred1) <3>fee = 0.0
   else {
      <4>_pred2 = !(car_type == "CAR" || car_type == "TRUCK")
      if (_pred2) <5>fee = -2.0 // INVALID
      else {
         <6>day, <7>duration = args[1], args[2]
         <8>_pred3 = car_type == "CAR"
         if (_pred3) <9>cost = compute_car_fee(duration)
         else <10>cost = compute_truck_fee(duration)
         <11>_pred4 = cost == -1.0 // EXCEED MAX DURATION
         if (_pred4) <12>fee = -1.0
         else {
            <13>_pred5 = day == "THURSDAY"
            if (_pred5) <14>cost = cost * THURSDAY_DISCOUNT
            else {
               <15> _pred6 = day == "SATURDAY"
               if (_pred6) <16>cost = cost * SATURDAY_SURCHARGE
            }
            <17>fee = cost
         ...}}}} // END of main
```

Figure 14: Pseudo-code of B&T

From the case study, we posit that the capability to focus on different bands of NDE can help an engineer better understand the code.

Note that Figure 13c also shows some false dependence edges in the CPDM. In the ground truth PDG, ⟨4⟩ affects ⟨11⟩, and ⟨11⟩ affects ⟨17⟩. However, since ⟨9-10⟩ and ⟨13⟩ always change their behaviors when ⟨4⟩ and ⟨11⟩ change, respectively, CPDM ignores the dependences $\{\langle 4\rangle\} \rightarrow \{\langle 11\rangle\}$ and $\{\langle 11\rangle\} \rightarrow \{\langle 17\rangle\}$ and adds $\{\langle 13\rangle\} \rightarrow \{\langle 17\rangle\}$ instead.

Per element inspection

We next focus on individual elements of the CPDM and their dependences. The difference in NDE expresses the detailed semantics around the element that conventional dependence analysis misses.

First, we investigate if CPDA can overcome a key limitation of the assumed transitivity inherent in static dependence analysis. Figure 15 shows the pseudo-code for mbe and mug, two small programs that explore the limits of static analysis. Each program includes program elements that are not dependent on one another, yet transitive static analysis is unable to realize this. In mbe, the key observation is that the value of `j` at ⟨7⟩ in any terminating execution of the program is independent of the value of `k`, as the loop termination condition depends only on `j`. However, from a dependence point of view the value of `k` assigned to nodes ⟨2⟩, ⟨5⟩, and ⟨6⟩ affects (via a data-dependence) the predicate `q(k)` of ⟨4⟩, on which ⟨7⟩ is control dependent. Thus, any transitive static dependence analysis will conclude that the value of `k` affects the value of `j`.

Journal Pre-proof

```
1  int mbe(int <1>j, int <2>k) {
2     while (<3>p(j)) {
3        if (<4>q(k)) {
4           <5>k = f1(k);
5        } else {
6           <6>k = f2(k);
7           <7>j = f3(j);
8        }
9     }
10    return <8>j;
11 }
```

(a) mbe

```
1  int mug(int <1>i, int <2>c, int
       <3>x) {
2     while (<4>p(i)) {
3        if (<5>q(c)) {
4           <6>x = f();
5           <7>c = g();
6        }
7        <8>i = h(i);
8     }
9     return <9>x;
10 }
11
```

(b) mug

Figure 15: Pseudo-code of (a) mbe and (b) mug

Similarly, in mug example the final value of `x` at $\langle 9 \rangle$ is independent of $\langle 7 \rangle$: if the initial value of `c` makes `q(c)` ($\langle 5 \rangle$) False, the variable `x` maintains its initial value Otherwise, if the predicate is True (one or more times) the return value of `g()` becomes the final value of `x`. However static dependence analysis finds that $\langle 7 \rangle$ affects $\langle 9 \rangle$ through path $\langle 7 \rangle \xrightarrow{d} \langle 5 \rangle \xrightarrow{c} \langle 6 \rangle \xrightarrow{d} \langle 9 \rangle$.[8]

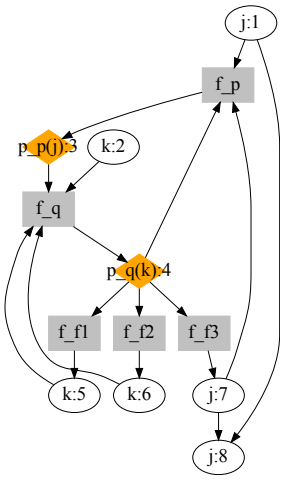
Figure 16a and 16b show the causal structures generated for mbe and mug, respectively. CPDA successfully discovers structures identical to the ground truth PDG for mbe and mug. What is different from the static dependence analysis is shown in Figure 16c, the ACE from other nodes to the return node for mbe and mug. The left side of the table shows no causal dependencies from nodes corresponding to variable `k` to $\langle 8 \rangle$. Nodes $\langle 1 \rangle$ or $\langle 7 \rangle$ are likewise devoid of dependence on variable `k`. The quantified dependence clearly captures that a change in the behavior of `k` has not affected the behavior of `j`, assuming that the program terminates. The right side of the table shows CPDA determines that $\langle 7 \rangle$ has no effect on $\langle 9 \rangle$, as changing $\langle 7 \rangle$ does not make a difference to the return value. These results demonstrate that CPDA can untangle the runtime dependence of the program. Unlike CPDA, PPDG and BNPDG cannot untangle the runtime dependence since they quantify the frequency of control/data-flow transition, which is insufficient to notice the behavior change. MOAD can untangle the runtime dependence as CPDA does, but it cannot produce the dependence graph like Figures 16a and 16b.

Next, we inspect how the difference in the magnitude of dependence exposes the detailed semantics of the program. To do so, we demonstrate how the magnitude of NDEs between the nodes explains the detailed dependence of the program for wc and tri. From the CPDA of wc, we observe the following (values in the parentheses are the degree of NDE):

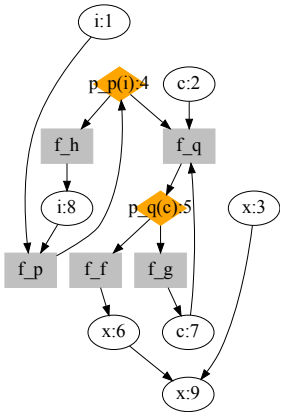
- $\{\langle 17 \rangle\} \rightarrow \{\langle 10 \rangle\}$ (1.00) $>$ $\{\langle 18 \rangle\} \rightarrow \{\langle 10 \rangle\}$ (0.80): In function `isLetter`, $\langle 17 \rangle$

[8] $\xrightarrow{d}$ and $\xrightarrow{c}$ denote the data- and control-dependency, respectively.

Journal Pre-proof



(a) mbe



(b) mbe

mbe		mug	
X	ACE ⟨X⟩ ↛ ⟨j:8⟩	X	ACE ⟨X⟩ ↛ ⟨x:9⟩
j:1	0.98	i:1	0.05
k:2	–	c:2	0.34
p:3	0.63	x:3	0.35
p:4	–	p:4	0.37
k:5	–	p:5	0.49
k:6	–	x:6	0.4
j:7	0.33	c:7	–
		i:8	–

(c) ACE

Figure 16: Causal structure of (a) mbe and (b) mug and the (c) causal dependency. In the graphs, the orange-diamond nodes represent the predicates in the program, and the grey-rectangle nodes represent the function invocations and their return values.

and ⟨18⟩ are the nodes that express whether the character is an alphabetic character or not. Since half of the tests lack non-alphabet characters, the effect on ⟨10⟩ of ⟨17⟩ is stronger than that of ⟨18⟩.

- $\{\langle 4\rangle\} \to \{\langle 11\rangle\}$ (1.00) > $\{\langle 13\rangle\} \to \{\langle 11\rangle\}$ (0.97) > $\{\langle 14\rangle\} \to \{\langle 11\rangle\}$ (0.77): ⟨4⟩, ⟨13⟩, and ⟨14⟩ all correspond to the variable `inword`. Because ⟨4⟩ affects ⟨11⟩ whenever there is at least one alphabetic character, $\{\langle 4\rangle\} \to \{\langle 11\rangle\}$ has the highest NDE, while ⟨13⟩ and ⟨14⟩ only affect ⟨11⟩ if the input includes more than one alphabetic character, which is true of fewer test cases. Furthermore, because there are test cases that include only alphabetic characters, ⟨13⟩ affects ⟨11⟩ more than ⟨14⟩ does.

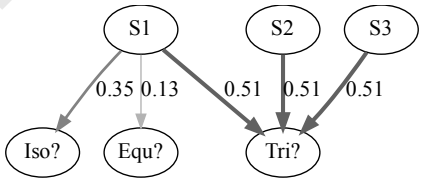


Figure 17: Partial graph from CPDM of tri

Figure 17 shows the partial graph from the CPDM of tri where Nodes ⟨S1⟩, ⟨S2⟩, ⟨S3⟩ represent inputs of the three sides lengths, and ⟨Tri?⟩, ⟨Equ?⟩, ⟨Iso?⟩ represent predicates checking whether the input is not-a-triangle, equilateral, or isosceles, respectively. In the CPDM, we observe the following:

Journal Pre-proof

- $\{\langle S1\rangle\} \rightarrow \{\langle Tri?\rangle\} = \{\langle S2\rangle\} \rightarrow \{\langle Tri?\rangle\} = \{\langle S3\rangle\} \rightarrow \{\langle Tri?\rangle\}$: The equivalence of these three weights in the CPDM is indicative of the symmetry in the use of the three side lengths for judging whether they form a triangle. While not shown in Figure 17, the CPDA also assigns undifferentiatable weights to dependences on $\langle Iso?\rangle$ and $\langle Equ?\rangle$. These examples show how the CPDM reveals the semantic symmetry beyond simple depends-on relations.

- $\{\langle S^*\rangle\} \rightarrow \{\langle Tri?\rangle\} > \{\langle S^*\rangle\} \rightarrow \{\langle Iso?\rangle\} > \{\langle S^*\rangle\} \rightarrow \{\langle Equ?\rangle\}$: In contrast, there is a clear difference in the dependence strength from a side length (e.g., $\langle S1\rangle$ in Figure 17) to the predicate nodes. This succinctly captures the relative challenge in finding inputs that affect each condition. For example, it is easier to meet the requirements of an isosceles triangle than an equilateral triangle.

  Note that frequency based approaches such as the PPDG and BNPDG cannot estimate such challenge: since all sides lengths always reach each predicate if executed, the PPDG and BNPDG consider the dependence between the side lengths and each predicate with the frequency of predicate execution. For example, if the nested if-structure in tri checks conditions in the order of $\langle Tri?\rangle$, $\langle Equ?\rangle$, and $\langle Iso?\rangle$, the PPDG/BNPDG will estimate $\{\langle S^*\rangle\} \rightarrow \{\langle Tri?\rangle\} > \{\langle S^*\rangle\} \rightarrow \{\langle Equ?\rangle\} > \{\langle S^*\rangle\} \rightarrow \{\langle Iso?\rangle\}$.

Summary of RQ2-1: By clustering strongly connected nodes based on the quantified dependence, CPDM can aid in grouping the program's functionality. Strong *NDE* values indicate dependency relations having an effect in most executions, such as the program's dominant control-flow structure. While focusing on a specific element, the relative *NDE* values demonstrate information on the local behavior of the element.

9.2.2. RQ2-2: Execution-awareness

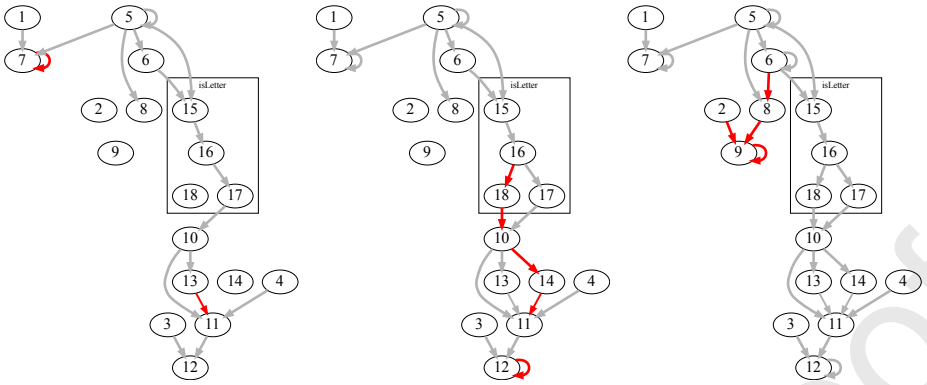
This section investigates how the CPDM changes when using different test suites.

Difference in the required functionality

Figure 18 shows *differences* in the resulting CPDMs for wc when using the four test suites introduced in Section 8.2. These test suites incrementally require additional functionality. The structure of the CPDM changes accordingly. In Figure 18, regarding the caption formatted as '$A - B$,' solid red edges are found only in the CPDM with the test suite A, and gray edges are found in both the CPDM with the test suites A and the CPDM with the test suite B. We actually represent edges only in the CPDM from the test suite B as dashed blue edges, but there are no such edges. We now consider the three comparisons shown in the figure in greater detail.

- One (multi-characters) word vs. one char (Figure 18a): Variable `characters` of $\langle 7\rangle$ either increments the prior values of 0 from $\langle 1\rangle$ or itself. The second of these only occurs when there is more than one character in the input. Thus, the self-dependence of $\langle 7\rangle$ appears when using the ***oneword*** test suite but not

Journal Pre-proof



(a) *oneword − onechar* (b) *oneline − oneword* (c) *multiline − oneline*

Figure 18: Differences in the CPDMs for wc created using the four different partitions of the test suite. In each figure caption $A - B$, shows edges in A only in solid-red, and edges in both in gray.

when using the ***onechar*** test suite. Similarly, the predicate of ⟨11⟩ is either affected by the initial value of variable `inword` at ⟨4⟩ or the assigned value at ⟨13⟩ or ⟨14⟩. The dependence on ⟨11⟩ from ⟨13⟩ finally appears when more than one alphabet character exists, yet the dependence on ⟨11⟩ from ⟨14⟩ does not, as it needs a test suite with at least one non-alphabetic character.

- One (multi-word) line vs. one word (Figure 18b): The main change of the inputs in the ***oneline*** test suite compared to the inputs in the ***oneword*** test suite is the inclusion of non-alphabet characters that separate the words. This brings into play the the sequence of nodes ⟨16⟩ → ⟨18⟩ → ⟨10⟩ → ⟨14⟩ → ⟨11⟩, which determine whether the current character is in a word or not. Also, a self-dependence involving ⟨12⟩ appears, as there is more than one word to count. Note that, unlike Figure 13b, the degree of *NDE* of {⟨18⟩} → {⟨10⟩} and {⟨17⟩} → {⟨10⟩} are the same. This is because every test in ***oneline*** contains both alphabet and non-alphabet characters.

- Multiple lines vs. one line (Figure 18c): The presence of newline characters in the ***multiline*** test suite bring into play wc's line counting functionality captured by ⟨2⟩, ⟨6⟩, ⟨8⟩, and ⟨9⟩, which is not present in the ***oneline*** or ***onechar*** inputs.

To summarize, the CPDM allows an engineer to uncover different flow patterns within the code by varying the test suite. Using different test suites that differ only in some key feature, an engineer can understand the connections in the program between the elements supporting that feature using the CPDM.

Differences in the input distribution

Figure 19 shows select NDEs for tri using three different test suites: ***Total***, which contains all 125 tests, ***Valid***, which contains the inputs that satisfy the

Journal Pre-proof

Test suite	⟨Tri?⟩	⟨Equ?⟩	⟨Iso?⟩
Total	0.51	0.13	0.35
Valid	**0.97**	0.13	0.35

(a) {⟨S1⟩} → {⟨predicate node⟩}

Ordered	⟨S1⟩	⟨S2⟩	⟨S3⟩
⟨Equ?⟩	0.29	**0.18**	0.24
⟨Iso?⟩	0.21	**0.46**	0.33

(b) {⟨S*⟩} → {⟨Equ?⟩,⟨Iso?⟩}

Figure 19: (a) Select NDEs involving ⟨S1⟩ (the NDEs for ⟨S2⟩ and ⟨S3⟩ are essentially the same) from tri using all tests (***Total***) and those satisfying the triangle inequality (***Valid***). (b) Select NDEs obtained using the ***Ordered*** test suite.

triangle inequality, and ***Ordered***, which contains the inputs where S1 ≤ S2 ≤ S3. The following two examples consider the impact of the differences in their distribution.

- In Table 19a the NDE of {⟨S1⟩} → {⟨Tri?⟩} is considerably larger in ***Valid*** (0.97) than in ***Total*** (0.51), while it is almost identical for {⟨S1⟩} → {⟨Equ?⟩} and {⟨S1⟩} → {⟨Iso?⟩}. The difference clearly demonstrates the ease of violating the triangle inequality.

- Turning to Table 19b, when compared to side lengths ⟨S1⟩ and ⟨S3⟩, the middle-length side, ⟨S2⟩, shows a smaller NDE with the equilateral condition and a larger NDE with the isosceles condition. If sides are ordered, the triangle is equilateral if and only if S1 = S3, and, if not, changing S2 cannot form an equilateral. Thus, ⟨S2⟩ only affects ⟨Equ?⟩ when the triangle is already an equilateral, so it has an NDE. In contrast, if the triangle is isosceles, ⟨S2⟩ is always one of two sides of equal length, consequently, has the highest NDE among the three sides. Notice that reaching-dependence based methods, such as the PPDG and BNPDG, cannot distinguish between the effect of each side length.

Summary of RQ2-2: The CPDM is able to differentiate the dependence pattern from different executions. Such information can highlight a part of a program related to a particular execution. Also, the CPDM can identify different dependence structures and their corresponding test cases. Leveraging this, applying CPDA to appropriate test subsets can make the CPDM distinctive. CPDA can guide test clustering and may even expose shortcomings in the current test suite.

9.2.3. RQ2-3: Debugging

The final scenario considers the challenge of debugging. This scenario illustrates how CPDA, which gives finer granularity dependence information and differentiates it by execution, can aid an engineer while debugging faulty code.

Figure 20 shows tcas buggy Version 1, tcas-1. The fault is on Line 5, in which the boolean operator '>' should be '>=.' Assume that we aim to find the location of this defect given a set of passing tests and a single failing test. In the failing test the effect of the defect must propagate to an output; consequently,

Journal Pre-proof

```
1 bool Non_Crossing_Biased_Climb() {
2   upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
3   if (upward_preferred)
4     result = !(Own_Below_Threat()) || ((Own_Below_Threat())
5       && (!(Down_Separation > ALIM())));  // bug: > should be >=
6   else ...
7   return result;
8 } ...
```

Figure 20: Faulty code in TCAS buggy version 1 (tcas-1)

an orthodox approach, which aims to reduce the search space for the faulty program element, is to compute a dynamic slice using a failing test (Agrawal and Horgan, 1990). However, as mentioned in Section 1, the use of a dynamic slice may yield a significant number of fault candidates. The dynamic slice of tcas-1 (colored nodes in Figure 21) is a typical example. Dicing (Agrawal et al., 1995) reduces the large candidate set by filtering out program elements in the dynamic slice of a passing execution. Yet, the defect may exist in both slices, as it does in tcas-1. Consequently, dicing will miss such defects.

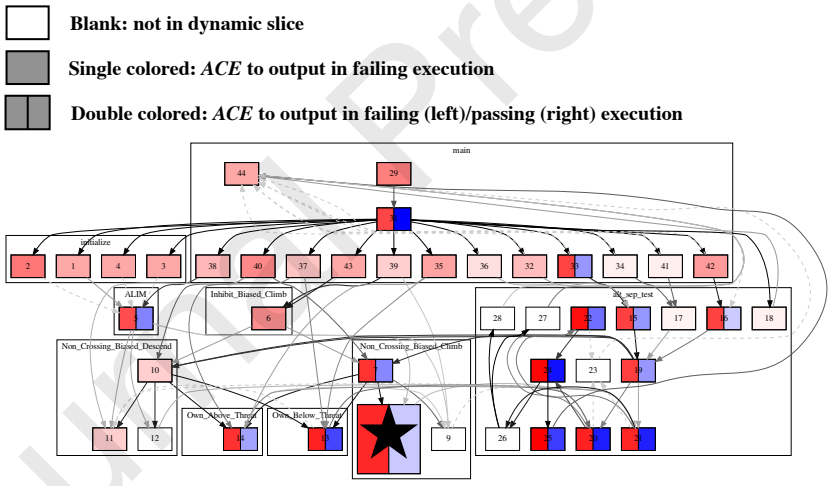


Figure 21: Illustration of the ACEs found for tcas-1. The darker the color, the stronger the ACE.

Where a binary decision is often too coarse, CPDA can quantify the (total) effect of a program element on an output (element). In Figure 21 the output has an ACE on the colored nodes where the darker the color, the stronger the dependence. Red is used for the ACE based on the failing test, while blue is used for the passing tests. Assuming that the defect strongly affects the failing output, we can reduce the number of candidates using the degree of the ACE. In

Journal Pre-proof

the case of tcas-1, only 15 nodes (double-colored in Figure 21) of the dynamic slices' 37 nodes have more than 0.5 ACE (41% reduction). While the search space has decreased considerably, we can narrow down the defect by employing a similar tactic to dicing, i.e., assuming that the defect has less effect on passing tests. The (darkness of the) blue color on the right side of the double-colored nodes in Figure 21 presents the degree of ACE of the 15 defect candidates to the passing output. Among them, $\langle\star\rangle$ has the smallest effect on the passing output and is the faulty node (`result`) in tcas-1.

The debugging scenario for tcas-1 illustrates how effective causal inference's quantified dependence can be at reducing debugging effort. In contrast, because the passing and failing tests follow almost identical control-flow, frequency based techniques such as the PPDG and BNPDG are ineffective at the same task: in PPDG, the probability of the reaching definition of the defect is high in the passing executions, while all the program elements have the same probability given the state of erroneous output in BNPDG.

Summary of RQ2-3: CPDA provides a finer granularity of dependence information than existing dependence analyses, which is an asset to the debugging process.

10. Threats to Validity

Using a limited set of program inputs and approximating dependence is an internal threat to any dynamic analysis and, thus, also to CPDA. However, CPDA is less vulnerable to the limited number of inputs than dynamic slicing techniques as its intervention relies not only on inputs but also on various mutations. In turn, CPDA's mutant sampling poses another threat to internal validity. The analysis may be biased if the mutation values are not representative of the actual program behavior. To mitigate this, we aim to choose the mutation values that are likely to be observed in real-world programs; if the value domain is known, we choose the mutation values from the domain. Otherwise, we use the distribution of the observed values in the test suite. To mitigate the selection bias, we use a sufficiently large number of samples based on the result of a preliminary experiment regarding the effect of N_{mpn} (Section 8.4). To choose a sufficient N_{mpn} to get steady empirical results, our preliminary experiment repeated the analysis while varying N_{mpn} (the number of samples) from 2 to 20 at intervals of two as well as 30, 40, 50. The results converged to similar values when $N_{mpn} \geq 10$. Based on this, we use a sufficiently large sample of 100 mutations ($N_{mpn} = 100$) for two smaller programs, tri and wc, and 20 mutations ($N_{mpn} = 20$) for B&T and tcas.

The use of six example programs with a single buggy version from the Siemens suite poses a threat to external validity. While our future plans include the intensive empirical study of CPDA and CPDM, our goal in this paper is to introduce the basic idea of causal dependence analysis and empirically validate that it is worthy of future study. Overall the results are quite promising. For example, we repeatedly found that the CPDM matched the basic structure

Journal Pre-proof

of the ground truth PDG despite performing no data or control dependence analysis. Thus we see the initial qualitative analysis presented in this paper as illustrative of our technique's future potential.

11. Related Work

This paper introduces Causal Program Dependence Analysis (CPDA) and empirically consider its use in building a Causal Program Dependence Model (CPDM). We thus consider work related to program dependence analysis. Because we study non-binary dependence weightings, we also consider work relating to probabilistic symbolic execution.

11.1. 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 (Ferrante et al., 1987), and has subsequently found many uses including program slicing (Horwitz and Reps, 1992; Horwitz et al., 1988). Dynamic dependence analysis incorporates one or more program inputs. A simple example is early dynamic slicing algorithms that computed a static slice of the PDG and then removed edges that were not executed (Agrawal and Horgan, 1990).

There have been a few proposals to quantify dependences, two of which we consider in our study: the PPDG (Baah et al., 2010) and the BNPDG (Yu et al., 2017). Both rely on frequency-of-execution to quantify dependence while avoiding any confounding bias by only considering the definitions that reach each variable during execution, ensuring that the quantified dependence reflects causation. However, they pay the cost of exact dataflow analysis. In contrast, CPDA is not tied to a statically computed PDG, freeing us from having to solve hard data-flow problems (such as pointer analysis) while enabling us to perform interprocedural analysis (neither the PPDG nor the BNPDG support interprocedural analysis). CPDA can also analyze purely value-centric dependence, while PPDG and BNPDG only detect those manifested by control-flow changes. A typical example involves the use of the modulo operator. For example given the expression `z = x + y % 2`, neither the PPDG nor the BNPDG distinguish the effect of the reaching definitions of `x` and `y`. In contrast, CPDA quantifies that `x` has a larger effect on `z` than `y` since the effect of changing `y` is partially masked by the modulo operation.

MOAD is another recent technique that estimates the degree of dependence without any static analysis by employing an observation-based method (Lee et al., 2019a; Lee et al., 2021). While, unlike PPDG and BNPDG, MOAD is able to quantify the effect in terms of the value difference as CPDA can, it cannot discriminate whether the effect is happening directly or indirectly, being incapable of producing a dependence graph.

Journal Pre-proof

Causal Program Slicing (Gore and Reynolds, 2009) (CPS) is another technique that aims to reason the program dependence and quantify the degree of dependence. While it also uses causal inference, the main purpose of CPS is on program slicing rather than a general program dependence analysis, as in CPDA. Consequently, it requires the static dependence graph, while CPDA does not. In addition, CPS gathers observations only from the input change, which is not as fine-grained as CPDA, as it can intervene at any program point and observe the effect of the change.

11.2. Probabilistic Symbolic Execution

Probabilistic Symbolic Execution (PSE) is another area that analyzes program semantics from a probabilistic perspective (Geldenhuys et al., 2012). PSE aims to ascertain how many inputs satisfy a particular path condition in a program. Given a symbolic path constraint and an input space, PSE uses a model counting technique (De Loera et al., 2004) to compute the ratio of inputs satisfying the condition. While PSE was initially restricted to solving linear constraints, more recent work adopted path decomposition and statistical approaches to handle arbitrarily complex mathematical constraints (Borges et al., 2015; Saha et al., 2022). The feasibility and scalability of PSE are nonetheless restricted by the computational cost of both symbolic execution and model counting. PSE is often unsuccessful in supporting non-linear constraints, sophisticated string operators, floating-point arithmetic, and inter-procedural analysis. An analytical approach is also typically incapable of analyzing heterogeneous features beyond the language's formal semantics, including incorporating third-party libraries or server-client communication. In addition, representing the counterfactual statement using a symbolic constraint requires a joint constraint of symbolic paths, whose computational cost can be significantly more expensive than the normal symbolic path. Recent work by Lee and Böhme (2023) exposes the limitation of symbolic execution and model counting-based probabilistic program analysis in terms of scalability and precision. In that same work, the authors propose a sampling-based statistical approach to overcome the limitation of analytical approaches.

11.3. Fault Localization

One application for CPDA is fault localization (FL). We have demonstrated how the ACE can facilitate finding the location of the fault. In general, CPDA differs from FL techniques in that the primary purpose of CPDA is to identify and understand the dependencies (including their structure) in any software system, while FL focuses specifically on pinpointing the location of the fault in the source code. More specifically, while FL and CPDA both rely on test inputs, unlike FL, CPDA does not require a test oracle[9], which provides the correctness of the program's output for each input. Our work also incorporates substantial

[9]The tests in CPDA are only used to observe the reference, i.e., the original behavior of the program.

Journal Pre-proof

case studies to showcase the effectiveness of causal dependencies of CPDA in other downstream activities.

One of the most widely studied FL techniques is Spectrum Based Fault Localization (SBFL), a dynamic approach that ranks program elements based on their suspiciousness, which is computed from test coverage and outcomes (Steimann et al., 2013). SBFL has been widely studied both as an independent technique (Abreu et al., 2007; Xie et al., 2013; Naish et al., 2011) and in hybridization with other FL input features and techniques (Sohn and Yoo, 2017; B. Le et al., 2016; Li et al., 2019; Le et al., 2015). However, it tends to produce many ties when program elements 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 (Masri and Assi, 2010).

Finally, several existing works utilize causal inference for fault localization. Baah et al. (2010, 2011) use a linear model to capture the causal effect from coverage of program elements to test outcomes. Gore and Reynolds (2012) and Shu et al. (2013) apply a similar linear regression approach to predicate values and method level coverage, respectively.

12. Discussion and Future Work

This section considers how more advanced causal models can improve CPDA. We also propose potential applications of CPDA/CPDM to other software engineering tasks.

Advanced Modeling

Our CPDA model considers the quantification of the program dependence in terms of how often one program element affects another. We choose the rate of occurrence in our initial model for two reasons. First, the observation for the causal analysis is sufficient to capture the existence of the behavior change. Second, it allows easy handling of complex data structures, such as trees. Yet, as we mentioned in Section 4, there could be many different notions regarding the degree of dependence. For instance, one might wish to quantify the strength of dependence regarding the magnitude of the change in the value of a variable. The question one wants to ask CPDA, then, would be something similar to "when there is an effect caused by element A on element B, how large/dramatic does it tend to be." Such a question could be more interesting than the question "does A affect B." For example, if one is interested in the program implementing "Newton's law of gravity",

$$F \sim \frac{M1 \cdot M2}{r^2},$$

the rate of occurrence of the dependence is not very interesting, as it always happens, but the magnitude of the change in the value of the variable F due to the other variables is more tempting to investigate. In such a case, the ratio of the change in the value of the source variable to the change in the value of the target variable could be used as a measure of dependence. A causal model

Journal Pre-proof

built for a lower-level representation of the program, such as bytecode, may allow us to quantify the magnitudes of changes for programs with complex data structures. Alternatively, future work will explore if general notions of similarity and diversity (Feldt et al., 2008, 2016) can be used to quantify levels of change and, thus, further refine CPDA modeling as well as the subsequent comprehension tasks.

There are two future technical improvements to CPDA. First, one of the existing challenges for causal inference that CPDA inherits is cycle handling. While causal inference was initially developed on acyclic structures, numerous scientific fields have attempted to extend causal inference to cyclic dependences. For example, a recent brain connectivity study considers variables in a temporal dimension to cater to cyclic relations (Chicharro and Panzeri, 2014). While the contextual constraints in programs prohibit such a temporal approach, the progress on the cyclic causal models (Rantanen et al., 2020b) suggests that CPDA will handle cycles better in the future.

The second aspect is to distinguish between control-dependence and data-dependence edges and to treat them separately. Distinguishing between control and data dependence can serve as additional information to understand the program's semantics. For example, in optimization and parallelization, treating data dependence separately assists in designing synchronization mechanisms that prevent race conditions and ensure data consistency (Lazarescu and Lavagno, 2012; Ketterlin and Clauss, 2012). Additionally, code obfuscation can help hide the control flow of the program, making it harder to reverse-engineer. Thus, each type of dependence finds independent uses in downstream software engineering tasks. In addition to the above, distinguishing may also help to reduce the imprecision of the discovered causal structure compared to the dependence graph in terms of the program semantics. As we mentioned in the result of RQ 1-1, one of the reasons for the imprecision is the masking effect of two independent dependencies, which can be compensated for by distinguishing between control and data dependence edges. One way to determine whether a dependence edge is a control or a data dependence is to check the source node of the edge; if the source node is a control node (e.g., a predicate), then the edge is a control dependence edge; otherwise, it is a data dependence edge.

Finally, we postulate that advanced modeling techniques can help to reduce the cost of intervention. If we use algorithms that can work from observational data only, the cost of CPDA can be significantly reduced. For example, we may be able to leverage observational data from existing test automation and regression testing activities that are typically available in modern Continuous Integration. We note that newer causal inference techniques can work with a combination of observational and interventional data (Guo et al., 2020): such methods have the potential to guide CPDA to adaptively intervene on program elements that are likely to lead to a relatively larger improvement in the dependence model, in a way similar to active learning (Cohn et al., 1996). In addition, the traces from the executions of different test cases themselves can be more efficiently obtained using advanced computation techniques such as incremental computation (Liu, 2023).

Journal Pre-proof

Potential Applications

Our program comprehension scenarios exploit the execution awareness of CPDA to extract information related to program semantics using known test inputs: by reversing this process, we posit that it is possible to extract information about test inputs using CPDA. Measuring and reporting quantified dependence relationships that come into existence during executions of different test cases may provide much richer information than binary coverage. Distances between CPDMs derived from different test suites can also provide valuable information to tasks such as test prioritization (Leon and Podgurski, 2003), failure clustering (Podgurski et al., 2003; Liu and Han, 2006), and scenario-based specification mining (Lo et al., 2007), in addition to the traditional distance metrics defined over coverage (Rothermel et al., 2001), test history (Hemmati et al., 2017), or lexical similarity (Thomas et al., 2014).

Our application of CPDA to debugging suggests that quantified dependence can potentially make a significant contribution to downstream maintenance tasks. In particular, we plan to investigate the notion of the *counterfactual* in the context of various mutation-based techniques for Automated Program Repair (APR) and Genetic Improvement (GI). In such applications, the question of *"what would have happened in a particular execution if a specific program element changed?"* plays a critical role.

One of the primary aspects of purely observation-based approach is handling programs written in languages with non-conventional semantics. Earlier work on ORBS successfully sliced Simulink/Stateflow models that are saved textually as an XML file (Gold et al., 2017) and images written in Picture Description Languages (PDLs) (Yoo et al., 2014, 2017). Likewise, estimating dependence relations and their degree employing non-conventional semantics of those languages is worth addressing. The central question is "how can we define the behavior of a program element?" While it is rather straightforward to consider the value (or trajectory) of a variable as the behavior of the program element, it is less clear for programs with non-conventional semantics such as those written using PDLs, where the individual effect of each program element is difficult to ascertain when focusing solely on the output. Subsequently, defining a mutation that well mimics the change of the program behavior also involves significant design choices. For instance, to mutate a program written in a dataflow programming language, either tweaking a single data packet, polluting the entire stream of data going out from a channel, or any intervention level between two are all feasible candidates for the mutation.

13. Conclusion

We propose Causal Program Dependence Analysis (CPDA), a way of identifying and then measuring the strength of the dependences between program elements by modeling their causal relationships. Existing dependence analysis techniques typically present binary relationships between program elements, ignoring the varying strengths of dependence relationships. By applying causal

Journal Pre-proof

inference to observational data from mutated executions, we quantify the degree of a value change in a program element A causing a value change in another program element B. Furthermore, we do this without the burden of static analysis. The paper also examines the benefit of quantified program dependence from CPDA on multiple applications, including a new graphical program dependence model Causal Program Dependence Model (CPDM). Our empirical results show that CPDA with its quantified dependence can aid engineers by identifying program semantics that would be missed when using conventional analyses.

References

Abreu, R., Zoeteweij, P., van Gemund, A.J.C., 2007. On the accuracy of spectrum-based fault localization, in: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), pp. 89–98. doi:10.1109/TAIC.PART.2007.13.

Agrawal, H., Horgan, J.R., 1990. Dynamic program slicing. SIGPLAN Not. 25, 246–256. URL: http://doi.acm.org/10.1145/93548.93576, doi:10.1145/93548.93576.

Agrawal, H., Horgan, J.R., London, S., Wong, W.E., 1995. Fault localization using execution slices and dataflow tests, in: Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95, pp. 143–151. doi:10.1109/ISSRE.1995.497652.

B. Le, T.D., Lo, D., Le Goues, C., Grunske, L., 2016. A learning-to-rank based fault localization approach using likely invariants, in: Proceedings of the 25th International Symposium on Software Testing and Analysis, ACM, New York, NY, USA. pp. 177–188. URL: http://doi.acm.org/10.1145/2931037.2931049, doi:10.1145/2931037.2931049.

Baah, G., Podgurski, A., Harrold, M., 2010. Causal inference for statistical fault localization, pp. 73–84. doi:10.1145/1831708.1831717.

Baah, G.K., Podgurski, A., Harrold, M.J., 2010. The probabilistic program dependence graph and its application to fault diagnosis. IEEE Transactions on Software Engineering 36, 528–545. doi:10.1109/TSE.2009.87.

Baah, G.K., Podgurski, A., Harrold, M.J., 2011. Matching Test Cases for Effective Fault Localization. Technical Report. Georgia Institute of Technology.

Binkley, D., 1997. Semantics guided regression test cost reduction. IEEE Transactions on Software Engineering 23, 498–516. doi:10.1109/32.624306.

Binkley, D., Gold, N., Harman, M., Islam, S., Krinke, J., Yoo, S., 2014. ORBS: Language-independent program slicing, in: Proceedings of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering, pp. 109–120.

Journal Pre-proof

Binkley, D., Gold, N., Harman, M., Islam, S., Krinke, J., Yoo, S., 2015. Orbs and the limits of static slicing, in: 2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 1–10. doi:10.1109/SCAM.2015.7335396.

Borges, M., Filieri, A., D'Amorim, M., Păsăreanu, C.S., 2015. Iterative distribution-aware sampling for probabilistic symbolic execution, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. pp. 866–877. URL: https://doi.org/10.1145/2786805.2786832, doi:10.1145/2786805.2786832.

Caillet, P., Klemm, S., Ducher, M., Aussem, A., Schott, A.M., 2015. Hip fracture in the elderly: a re-analysis of the epidos study with causal bayesian networks. PLoS One 10, e0120125.

Chicharro, D., Panzeri, S., 2014. Algorithms of causal inference for the analysis of effective connectivity among brain regions. Frontiers in neuroinformatics 8, 64.

Cohn, D.A., Ghahramani, Z., Jordan, M.I., 1996. Active learning with statistical models. Journal of artificial intelligence research 4, 129–145.

Collard, M., Decker, M., Maletic, J., 2013. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration, pp. 516–519. doi:10.1109/ICSM.2013.85.

De Loera, J.A., Hemmecke, R., Tauzer, J., Yoshida, R., 2004. Effective lattice point counting in rational convex polytopes. Journal of Symbolic Computation 38, 1273–1302. URL: https://www.sciencedirect.com/science/article/pii/S0747717104000422, doi:https://doi.org/10.1016/j.jsc.2003.04.003. symbolic Computation in Algebra and Geometry.

Do, H., Elbaum, S., Rothermel, G., 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Softw. Engg. 10, 405–435. URL: https://doi.org/10.1007/s10664-005-3861-2, doi:10.1007/s10664-005-3861-2.

Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D., 2001. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27, 99–123. doi:10.1109/32.908957.

Ettinger, R., Verbaere, M., 2004. Untangling: A slice extraction refactoring, in: Proceedings of the 3rd International Conference on Aspect-oriented Software Development, ACM, New York, NY, USA. pp. 93–101. URL: http://doi.acm.org/10.1145/976270.976283, doi:10.1145/976270.976283.

Feldt, R., Poulding, S., 2013. Finding test data with specific properties via metaheuristic search, in: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE), IEEE. pp. 350–359.

Journal Pre-proof

Feldt, R., Poulding, S., Clark, D., Yoo, S., 2016. Test set diameter: Quantifying the diversity of sets of test cases, in: 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 223–233. doi:10.1109/ICST.2016.33.

Feldt, R., Torkar, R., Gorschek, T., Afzal, W., 2008. Searching for cognitively diverse tests: Towards universal test diversity metrics, in: 2018 Advances in Neural Information Processing Systems, IEEE. pp. 178–186.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization 9, 319–349.

Gallagher, K.B., Lyle, J.R., 1991. Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17, 751–761. doi:10.1109/32.83912.

Geldenhuys, J., Dwyer, M.B., Visser, W., 2012. Probabilistic symbolic execution, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA. pp. 166–176. URL: https://doi.org/10.1145/2338965.2336773, doi:10.1145/2338965.2336773.

Gold, N.E., Binkley, D., Harman, M., Islam, S., Krinke, J., Yoo, S., 2017. Generalized observational slicing for tree-represented modelling languages, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA. pp. 547–558. URL: http://doi.acm.org/10.1145/3106237.3106304, doi:10.1145/3106237.3106304.

Gore, R., Reynolds, P.F., 2009. Causal program slicing. 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation , 19–26.

Gore, R., Reynolds, Jr., P.F., 2012. Reducing confounding bias in predicate-level statistical debugging metrics, in: Proceedings of the 34th International Conference on Software Engineering, IEEE Press. pp. 463–473.

Guo, R., Cheng, L., Li, J., Hahn, P.R., Liu, H., 2020. A survey of learning causality with data: Problems and methods. ACM Computing Surveys (CSUR) 53, 1–37.

Hemmati, H., Fang, Z., Mäntylä, M.V., Adams, B., 2017. Prioritizing manual test cases in rapid release environments. Software Testing, Verification and Reliability 27, e1609.

Holland, P.W., 1988. Causal inference, path analysis and recursive structural equations models. ETS Research Report Series 1988, i–50.

Horwitz, S., Reps, T., 1992. The use of program dependence graphs in software engineering, in: 14^{th} International Conference on Software Engineering, Melbourne, Australia. pp. 392–411.

Journal Pre-proof

Horwitz, S., Reps, T., Binkley, D., 1988. Interprocedural slicing using dependence graphs. SIGPLAN Not. 23, 35–46. URL: http://doi.acm.org/10.1145/960116.53994, doi:10.1145/960116.53994.

Jiang, S., McMillan, C., Santelices, R., 2017. Do programmers do change impact analysis in debugging? Empirical Software Engineering 22, 631–669. URL: https://doi.org/10.1007/s10664-016-9441-9, doi:10.1007/s10664-016-9441-9.

Karim, R., Tip, F., Sochurkova, A., Sen, K., 2018. Platform-independent dynamic taint analysis for javascript. IEEE Transactions on Software Engineering , 1–1doi:10.1109/TSE.2018.2878020.

Ketterlin, A., Clauss, P., 2012. Profiling data-dependence to assist parallelization: Framework, scope, and optimization, in: 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, pp. 437–448. doi:10.1109/MICRO.2012.47.

Lazarescu, M.T., Lavagno, L., 2012. Dynamic trace-based data dependency analysis for parallelization of c programs, in: 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation, pp. 126–131. doi:10.1109/SCAM.2012.15.

Le, T.D.B., Oentaryo, R.J., Lo, D., 2015. Information retrieval and spectrum based bug localization: Better together, in: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ACM, New York, NY, USA. pp. 579–590. URL: http://doi.acm.org/10.1145/2786805.2786880, doi:10.1145/2786805.2786880.

Lee, S., 2020. Scalable and approximate program dependence analysis, in: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings, Association for Computing Machinery, New York, NY, USA. pp. 162–165. URL: https://doi.org/10.1145/3377812.3381392, doi:10.1145/3377812.3381392.

Lee, S., Binkley, D., Feldt, R., Gold, N., Yoo, S., 2019a. Moad: Modeling observation-based approximate dependency, in: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 12–22. doi:10.1109/SCAM.2019.00011.

Lee, S., Binkley, D., Feldt, R., Gold, N., Yoo, S., 2021. Observation-based approximate dependency modeling and its use for program slicing. Journal of Systems and Software 179, 110988. URL: https://www.sciencedirect.com/science/article/pii/S0164121221000856, doi:https://doi.org/10.1016/j.jss.2021.110988.

Lee, S., Binkley, D., Gold, N., Islam, S., Krinke, J., Yoo, S., 2020. Evaluating lexical approximation of program dependence. Journal of Systems and Software 160, 110459. URL: http://www.sciencedirect.com/science/

Journal Pre-proof

article/pii/S016412121930233X, doi:https://doi.org/10.1016/j.jss.2019.110459.

Lee, S., Böhme, M., 2023. Statistical reachability analysis, in: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 326–337. URL: https://doi.org/10.1145/3611643.3616268, doi:10.1145/3611643.3616268.

Lee, S., Hong, S., Yi, J., Kim, T., Kim, C., Yoo, S., 2019b. Classifying false positive static checker alarms in continuous integration using convolutional neural networks, in: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 391–401. doi:10.1109/ICST.2019.00048.

Leon, D., Podgurski, A., 2003. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases, in: Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE 2003), IEEE Computer Society Press. pp. 442–456.

Li, X., Li, W., Zhang, Y., Zhang, L., 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization, in: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA. pp. 169–180.

Liu, C., Han, J., 2006. Failure proximity: a fault localization-based approach, in: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, pp. 46–56.

Liu, Y.A., 2023. Incremental computation: What is the essence? arXiv:2312.07946.

Lo, D., Maoz, S., Khoo, S.C., 2007. Mining modal scenario-based specifications from execution traces of reactive systems, in: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, Association for Computing Machinery, New York, NY, USA. p. 465–468. URL: https://doi.org/10.1145/1321631.1321710, doi:10.1145/1321631.1321710.

Masri, W., Assi, R., 2010. Cleansing test suites from coincidental correctness to enhance fault-localization, in: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on, pp. 165–174.

Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Transactions on Software Engineering Methodology 20, 11:1–11:32.

Pearl, J., 2001. Direct and indirect effects, in: Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. pp. 411–420.

Journal Pre-proof

Pearl, J., 2009. Causality. Cambridge University Press. doi:10.1017/CBO9780511803161.

Pearl, J., 2019. The seven tools of causal inference, with reflections on machine learning. Communications of the ACM 62, 54–60.

Pearl, J., et al., 2009. Causal inference in statistics: An overview. Statistics surveys 3, 96–146.

Peters, J., Janzing, D., Schölkopf, B., 2017. Elements of causal inference: foundations and learning algorithms. MIT press.

Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., Wang, B., 2003. Automated support for classifying software failure reports, in: 25th International Conference on Software Engineering, 2003. Proceedings., pp. 465–475. doi:10.1109/ICSE.2003.1201224.

Ramalingam, G., 1994. The undecidability of aliasing. ACM Trans. Program. Lang. Syst. 16, 1467–1471.

Rantanen, K., Hyttinen, A., Järvisalo, M., 2018. Learning optimal causal graphs with exact search, in: International Conference on Probabilistic Graphical Models, PMLR. pp. 344–355.

Rantanen, K., Hyttinen, A., Järvisalo, M., 2020a. Discovering causal graphs with cycles and latent confounders: an exact branch-and-bound approach. International Journal of Approximate Reasoning 117, 29–49.

Rantanen, K., Hyttinen, A., Järvisalo, M., 2020b. Learning optimal cyclic causal graphs from interventional data.

Richens, J.G., Lee, C.M., Johri, S., 2020. Improving the accuracy of medical diagnosis with causal machine learning. Nature communications 11, 1–9.

Rothermel, G., Untch, R.H., Chengyun Chu, Harrold, M.J., 2001. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering 27, 929–948. doi:10.1109/32.962562.

Saha, S., Downing, M., Brennan, T., Bultan, T., 2022. Preach: A heuristic for probabilistic reachability to identify hard to reach statements, in: International Conference on Software Engineering (ICSE). doi:10.1145/3510003.3510227.

Scutari, M., Vitolo, C., Tucker, A., 2019. Learning bayesian networks from big data with greedy search: computational complexity and efficient implementation. Statistics and Computing 29, 1095–1108. URL: https://doi.org/10.1007/s11222-019-09857-1, doi:10.1007/s11222-019-09857-1.

Journal Pre-proof

Shu, G., Sun, B., Podgurski, A., Cao, F., 2013. Mfl: Method-level fault localization with causal inference, in: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, pp. 124–133. doi:10.1109/ICST.2013.31.

Singh, M., Valtorta, M., 1995. Construction of bayesian network structures from data: A brief survey and an efficient algorithm. International Journal of Approximate Reasoning 12, 111–131. URL: https://www.sciencedirect.com/science/article/pii/0888613X9400016V, doi:https://doi.org/10.1016/0888-613X(94)00016-V.

Sohn, J., Yoo, S., 2017. Fluccs: using code and change metrics to improve fault localization, pp. 273–283. doi:10.1145/3092703.3092717.

Spirtes, P., Zhang, K., 2016. Causal discovery and inference: concepts and recent methodological advances, in: Applied informatics, SpringerOpen. pp. 1–28.

Steimann, F., Frenkel, M., Abreu, R., 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis, ACM, New York, NY, USA. pp. 314–324.

Stroock, D.W., 2010. Probability theory: an analytic view. Cambridge university press.

Thomas, S.W., Hemmati, H., Hassan, A.E., Blostein, D., 2014. Static test case prioritization using topic models. Empirical Software Engineering 19, 182–212. URL: https://doi.org/10.1007/s10664-012-9219-7, doi:10.1007/s10664-012-9219-7.

Untch, R.H., Offutt, A.J., Harrold, M.J., 1993. Mutation analysis using mutant schemata. SIGSOFT Softw. Eng. Notes 18, 139–148. URL: https://doi.org/10.1145/174146.154265, doi:10.1145/174146.154265.

Xie, X., Chen, T.Y., Kuo, F.C., Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering Methodology 22, 31:1–31:40.

Yoo, S., Binkley, D., Eastman, R., 2014. Seeing is slicing: Observation based slicing of picture description languages, in: 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp. 175–184. doi:10.1109/SCAM.2014.26.

Yoo, S., Binkley, D., Eastman, R., 2017. Observational slicing based on visual semantics. Journal of Systems and Software 129, 60–78.

Yu, X., Liu, J., Yang, Z., Liu, X., 2017. The bayesian network based program dependence graph and its application to fault localization. Journal of Systems and Software 134, 44 – 53. URL: http://www.sciencedirect.com/science/article/pii/S0164121217301796, doi:https://doi.org/10.1016/j.jss.2017.08.025.

Journal Pre-proof

Yu, X., Liu, J., Yang, Z.J., Liu, X., Yin, X., Yi, S., 2016. Bayesian network based program dependence graph for fault localization, in: 2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 181–188. doi:10.1109/ISSREW.2016.35.

Zhifeng Yu, Rajlich, V., 2001. Hidden dependencies in program comprehension and change propagation, in: Proceedings 9th International Workshop on Program Comprehension. IWPC 2001, pp. 293–299. doi:10.1109/WPC.2001.921739.

Journal Pre-proof

Journal Pre-proof

Appendix A. Figures in RQ1-1

Below three figures shows the causal structure from *CP*-method and *HS*-method for three subjects B&T, wc, and tcas. The box represents the function boundary.

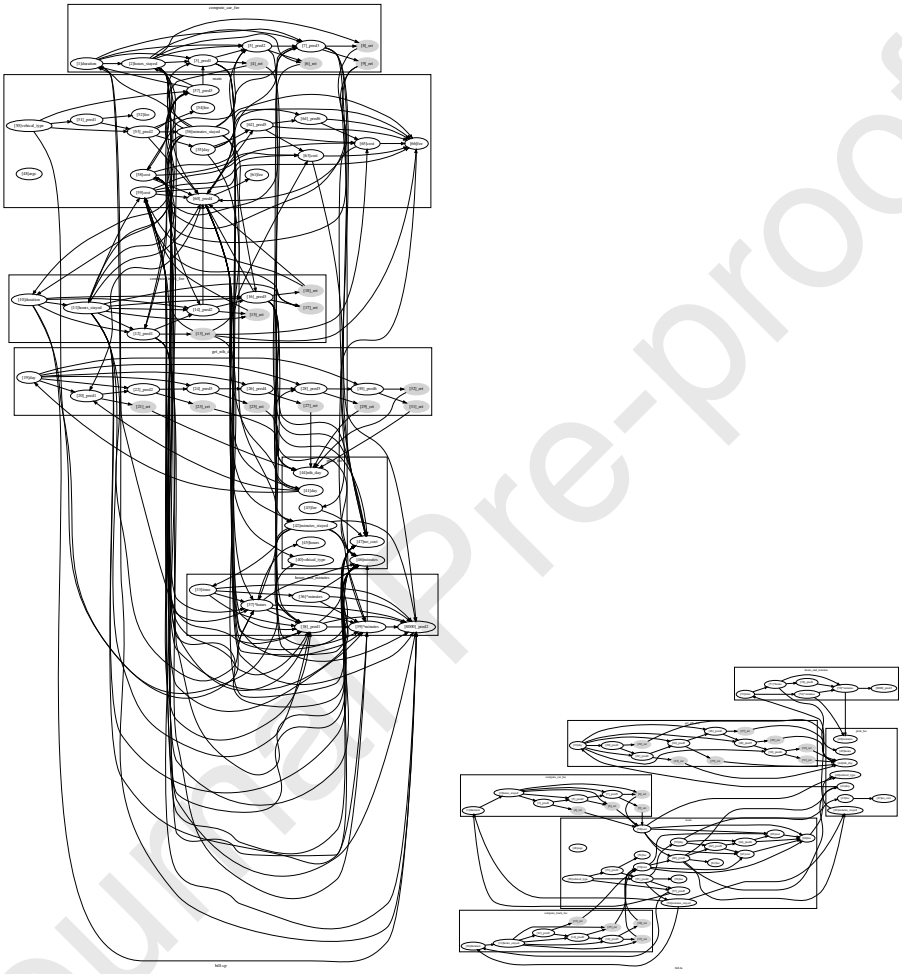


Figure A.22: Causal structure of B&T (left: the *CP*-method; right: the *HS*-method)

Journal Pre-proof

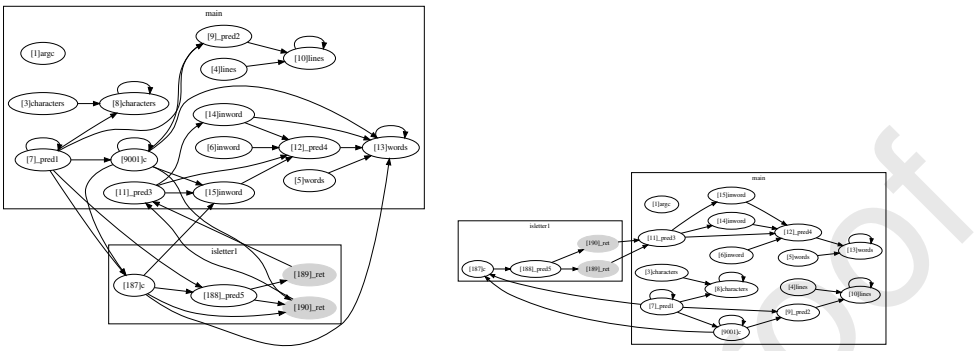


Figure A.23: Causal structure of wc (left: the *CP*-method; right: the *HS*-method)

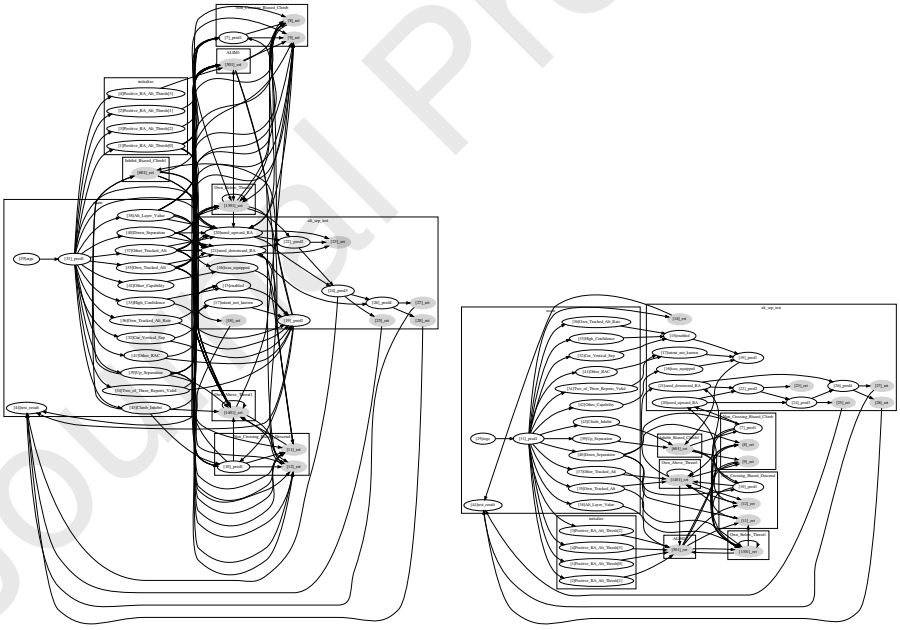


Figure A.24: Causal structure of tcas (left: the *CP*-method; right: the *HS*-method)

Journal Pre-proof

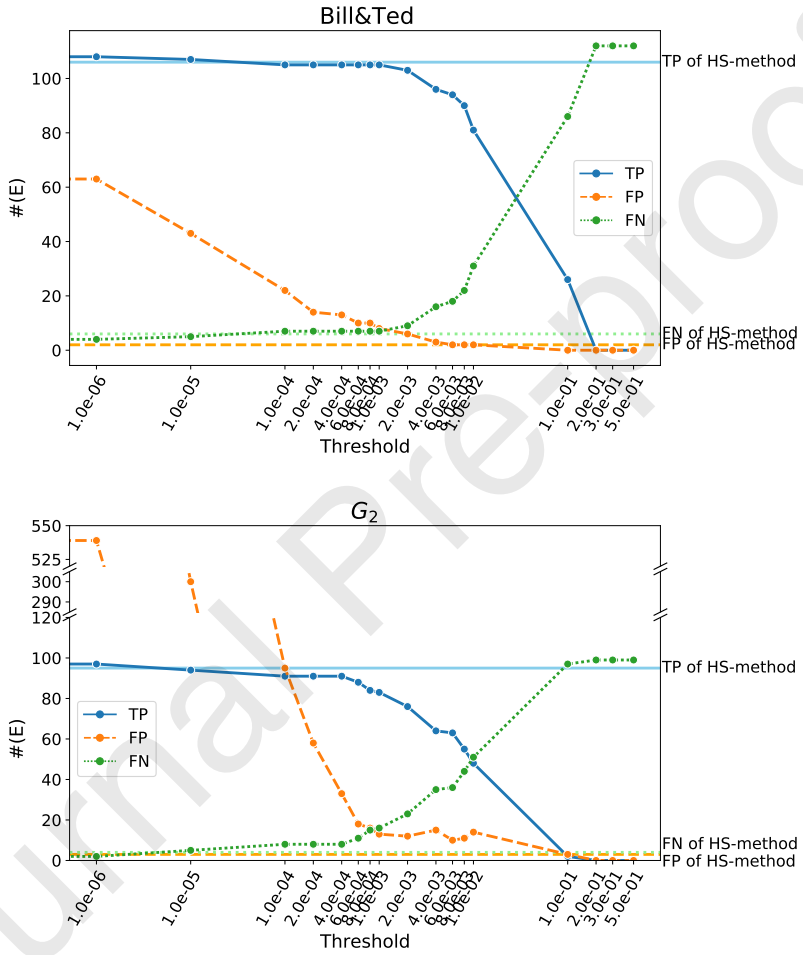


Figure A.25: The number of true-positive (TP), false-positive (FP), and false-negative (FN) edges found in the causal structure of the *RCP*-method using different thresholds. The horizontal lines shows TP, FP, and FN of the causal structure of the *HS*-method.

Journal Pre-proof

Declaration of interests

☐ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☒ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Shin Yoo reports financial support was provided by Korea Ministry of Science and ICT. Robert Feldt reports was provided by Swedish Scientific Council. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Journal Pre-proof