The Probabilistic Program Dependence Graph and Its Application to Fault Diagnosis

George K. Baah
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
baah@cc.gatech.edu

Andy Podgurski
Electrical Engineering and Computer Science Dept.
Case Western Reserve University
Cleveland, OH 44106
andy@eecs.case.edu

Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
harrold@cc.gatech.edu

ABSTRACT

This paper presents an innovative model of a program's internal behavior over a set of test inputs, called the probabilistic program dependence graph (PPDG), that facilitates probabilistic analysis and reasoning about uncertain program behavior, particularly that associated with faults. The PPDG is an augmentation of the structural dependences represented by a program dependence graph with estimates of statistical dependences between node states, which are computed from the test set. The PPDG is based on the established framework of probabilistic graphical models, which are widely used in applications such as medical diagnosis. This paper presents algorithms for constructing PPDGs and applying the PPDG to fault diagnosis. This paper also presents preliminary evidence indicating that PPDGs can facilitate fault localization and fault comprehension.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging—*Diagnostics, Monitors*

General Terms: Algorithms, Experimentation

Keywords: probabilistic graphical models, machine learning, fault diagnosis, program analysis

1. INTRODUCTION

A variety of graphical models have been used in software engineering applications to abstract relevant relationships between program elements or states, and thereby facilitate program analysis and understanding. These models include control-flow graphs, call graphs, finite-state automata, and program dependence graphs. Models produced by static analysis generally indicate that certain occurrences are *possible* at runtime (e.g., control transfers, calls, state occurrences, state transitions, and information flows), whereas models produced by dynamic analysis indicate what actually does occur during one or more executions. However, commonly used graphical models of internal program dynamics do not support making inferences about how likely particular program behaviors are. The inability of the models to make inferences about program behaviors severely limits the utility of the models for reasoning about the causes and effects of inherently uncertain program behaviors, such as runtime failures.

Program dependence graphs (PDGs) [7], which have proven useful in such software engineering applications as testing [11], debugging [28], and maintenance [9], model potential semantic dependences [23] between program elements. However, they do not model the strengths of any corresponding statistical dependences between the program elements. This paper makes the case that by augmenting PDGs with statistical dependence (and independence) information in the principled way provided by probabilistic graphical models [20], it is possible to substantially increase the utility of PDGs in some software engineering applications. Probabilistic graphical models have proven useful in several fields (e.g., medicine [8] and robotics [26]) due to their ability to model both the presence of certain dependences between variables of interest and the way in which the variables are probabilistically conditioned on other variables. A probabilistic graphical model derived from a PDG provides a natural framework for modeling both the presence of dependences and their statistical strengths.

In this paper, we present our technique that uses the program dependence graph to create a novel probabilistic graphical model. The model captures the conditional statistical dependence and independence relationships among program elements in a way that facilitates making probabilistic inferences about program behaviors. We call this model a *Probabilistic Program Dependence Graph* (PPDG). Our technique produces the PPDG for a program by augmenting its program dependence graph automatically. The technique retains the nodes and edges of the original PDG and, in some cases, adds nodes and edges. The technique associates a set of abstract states with each node in the PPDG. Each abstract state represents a (possibly large) set of concrete nodes states, in a way that is chosen to be relevant to one or more applications of PPDGs. Each node has a conditional probability distribution that relates the states of the node to the states of its parent nodes. The technique estimates the parameters of the probability distribution by analyzing executions of the program, which are induced by a set of test cases or captured program inputs.

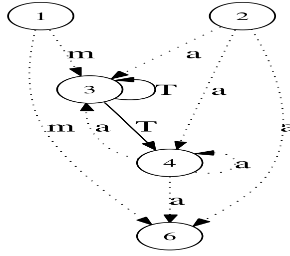
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA'08, July 20-24, 2008, Seattle, Washington, USA.

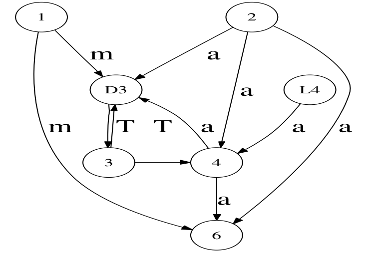
Copyright 2008 ACM 978-1-60558-050-0/08/07 ...$5.00.

```
   int Prog (){
1       int m = read();
2       int a = read();
3       while(a < m){
4           a++;
5       }
6       return m+a;
7  }
```

(a)



(b)



(c)

Figure 1: (a) Example program (Prog), (b) PDG, and (c) transformed PDG.

The ability of PPDGs to facilitate probabilistic reasoning about program behaviors makes them potentially valuable for several software engineering tasks. In this paper, we present preliminary evidence indicating that PPDGs can be useful for fault localization and fault comprehension. Intuitively, PPDGs are well suited to these tasks because they can indicate how a failing execution differs from successful ones, both structurally and statistically, and because context information can be generated from PPDGs that is useful for understanding why a particular program statement might be suspected of causing a given failure. More generally, a PPDG can be used as a knowledge base, which can be analyzed with different algorithms to understand various program behaviors.

The main contributions of this paper are:

- The Probabilistic Program Dependence Graph (PPDG), a novel probabilistic graphical model of program behavior based on the program dependence graph;
- Applications of the PPDG to fault localization and fault comprehension;
- The results of experiments and case studies, which indicate that the PPDG is potentially useful for these applications.

2. BACKGROUND

In this section, we briefly review program dependence graphs, which represent structural dependences between program statements, and define a type of probabilistic graphical model called a dependency network [12], which represents conditional dependence and independence relationships between random variables. These two types of models are the basis for the Probabilistic Program Dependence Graph.

2.1 Program Dependence Graph

A *program dependence graph* (PDG) [7] consists of nodes and directed edges, where nodes represent program statements and directed edges represent control or data dependences between the nodes. Informally, a node X is *control dependent* on node Y if Y represents the predicate of a conditional branch that directly controls whether X is executed. A node X is *data dependent* on node Y if a variable v defined at Y is used at X and there is a path of the form $Y \cdot P \cdot X$, where P is a path along which v is not redefined. Intuitively, P permits the value of v to flow from Y to X.

Figures 1(a) and 1(b) show an example program (`Prog`), and its corresponding program dependence graph, respectively. The nodes in the program dependence graph are labeled with the line numbers of the statements in the program. Solid edges represent control dependences between nodes and dotted edges represent data dependences between nodes. Labels on the control dependence edges are either "T" for true or "F" for false. Labels on the data dependence edges represent the variables involved in the data flow between the nodes.

For example, in Figure 1(b), node 4 is control dependent on node 3 and it is data dependent on itself and node 2. The control dependence edge between node 3 and node 4 has the label "T", which signifies that node 4 is executed when the branch condition at node 3 is true. The label on the data flow edge between node 4 and node 6 is "a," which implies that the value of variable a at node 4 flows to node 6.

2.2 Probabilistic Graphical Model

A *probabilistic graphical model* [20] is an annotated graph that captures the probabilistic relationships among a set of random variables. The nodes in the graph represent random variables and the arcs represent conditional dependences between the random variables. There are different kinds of probabilistic graphical models, including Bayesian networks [21], Markov random fields [20], and dependency networks [12]. Bayesian networks are directed acyclic graphs, whereas Markov random fields are undirected graphs. Dependency networks are similar to Bayesian networks except that the former may contain cycles.

DEFINITION 1. A *dependency network* is a triple (S, G, P) where S represents a set of random variables, $G = (N, E)$ is a possibly cyclic directed graph, and P represents a set of conditional probability distributions. N and E represent the set of nodes and directed edges in G, respectively, with nodes in G corresponding to random variables in S and edges in G representing dependences among the random variables.

Let $S = \{X_1, \cdots, X_n\}$ be a set of random variables. Each node in G corresponds to a variable $X_j \in S$ and directed edges between nodes represent dependences between the variables in S. Given a node in G, the parents of X_j ($Pa(X_j)$) are the nodes that render X_j conditionally independent[1] of the other variables. For graph G, there is a directed edge from each parent of X_j to X_j. Each X_j has a set of discrete states $x = \{x_1, \cdots, x_k\}$, and X_j can assume any of the states $x_i \in x$. Each node has a conditional probability distribution, $p(X_j|Pa(X_j)) \in P$, relating the states of X_j to the states of its parents $Pa(X_j)$.

Figure 2 shows an example of a dependency network that

[1]Two random variables A and B are said to be *conditionally independent* given C, if the distribution of A does not depend on B given that the value of C is known (i.e., $p(A|B, C) = p(A|C)$).

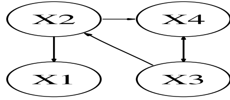


Figure 2: An example of a dependency network.

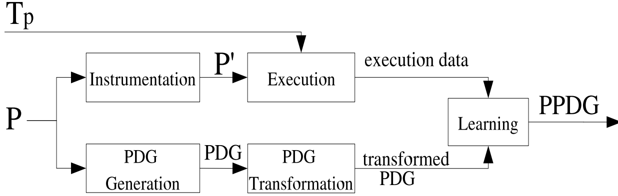


Figure 3: Building a PPDG.

has four random variables: X_1, X_2, X_3, and X_4. Thus, there are four conditional probability distributions for the network. The conditional probability distribution for X_1,X_2,X_3, and X_4 are $p(X_1|X_2)$, $p(X_2|X_3)$, $p(X_3|X_4)$, and $p(X_4|X_2, X_3)$ respectively.

3. PROBABILISTIC PROGRAM DEPENDENCE GRAPH

Our PPDG model is based on the dependency network model defined in Section 2. We use dependency networks because they permit directed cycles, which are present in the program dependence graphs of typical programs, due to the presence of program loops. Henceforth, we shall use the terms "loop" and "cycle" interchangeably.

The process of producing a PPDG consists of five main steps: PDG generation, PDG transformation, instrumentation, execution, and learning. These steps are illustrated in Figure 3. First, the PDG-generation step generates the PDG of the input program P. The PDG is then input into the PDG transformation step, which transforms the PDG, resulting in a transformed PDG. The instrumentation step inserts probes into P to gather the execution data needed to estimate the parameters of the PPDG, and produces the instrumented program P'. During the execution step, P' is executed with its test suite T_P to generate the execution data. Finally, the learning step generates a PPDG based on the execution data and transformed PDG.

Definition 2. *A Probabilistic Program Dependence Graph* (PPDG) for program P is a triple (G, S, Q) where $G = (N, E)$ is the transformed PDG of P, whose node and edge sets are N and E, respectively, and S and Q are mappings from nodes to states and nodes to conditional probability distributions, respectively, that relate the states of nodes N in G.

Figure 1 shows an example program (`Prog`) with its corresponding PDG and transformed PDG. We use the example program throughout our discussion of the steps involved in generating a PPDG.

3.1 PDG Transformation

During this step, our technique transforms the PDG by adding nodes to it and specifying the states of the nodes. The technique treats all nodes in the PDG as random variables; hence, we use the terms "nodes" and "random variables" interchangeably. The technique assigns to each node in a program's transformed PDG a finite set of discrete abstract states, each of which represents a set of related concrete states of the corresponding statement. Hereafter, we use the term "state" to refer to an abstract state. The states of a node must be *mutually exclusive*[2] (i.e., a node cannot be in two different states at the same time). Before a node is executed, it has the default state denoted by the symbol $\perp$. When a node is executed, it assumes a state distinct from $\perp$, and it is said to be an *active* state. Each node has a conditional probability distribution, that relates the states of the node to the states of its parent nodes in the PPDG. We call the graph that results from the PDG transformation step the *transformed PDG*. Figure 1(c) shows the structure of the transformed PDG of the example program (`Prog`).

3.1.1 Specifying Node States

The state of a PPDG node abstracts the part of a program's state that pertains to the node when the program executes. There are different ways to model this "local" concrete state. In this paper, our technique models the state in one or both of two simple ways depending on whether the node represents a branch predicate, a statement that uses one or more variables, or both. These characterizations are intended to reflect certain aspects of a node's concrete state that are relevant to applications such as fault localization and fault comprehension. (Other aspects are also relevant, but we shall not consider them in this paper.) Our technique characterizes the state of a node representing a branch predicate by the outcome of the predicate, and it characterizes the state of a node representing a statement s that uses one or more variables by the set of variable definitions that reach those uses during execution (i.e., by the definitions on which s is *dynamically data dependent*). Note that the static dependences represented in a program dependence graph do not in general reflect the dynamic data dependences accurately.

The state of a predicate node can be characterized by both a predicate outcome and a set of dynamic dependences. Thus, the state of a predicate node may have two components or substates (i.e., a predicate substate and a data dependence substate). If a node has two substates, our technique splits the node into two nodes and assigns a substate to each node. (We discuss substates further in Section 3.1.2.) In this paper, all predicates are transformed into simple predicates. A *simple predicate* is a predicate of the form ($v1$ *relop* $v2$) where $v1$ and $v2$ are program variables. Our technique assumes that all conditions with compound predicates (i.e., conjunctions and/or disjunctions of simple predicates) are transformed into conditions with simple predicates. Note that if a condition (e.g., $if(v1)$) consists of a single variable (i.e., $v1$), our technique treats the condition as $if(v1 == 0)$. Hence, the predicate for the condition is $v1 == 0$ (i.e., $v2 == 0$). Transforming all predicates into simple predicates simplifies the specification of node states.

[2]Note that our technique assumes it is processing the PDG of a sequential program—a concurrent program can cause a node to be in two states at the same time thus, violating the mutually-exclusive property of the states of nodes.

Modeling Predicate Outcomes

The types of predicate outcomes that occur at a node in a PDG determine the kinds of states that are associated with the node. For a conditional statement (e.g., *if-statement*), the state of its simple predicate characterizes the state of the conditional statement. The predicate outcomes depend on how the program variables involved in the predicate computation relate to each other in terms of the relational operators (i.e., $<$, $>$, $\leq$, $\geq$, $==$, and $\neq$). The technique places the simple predicates into two categories based on the state assignments.

1. For nodes whose predicates involve primitive variables [e.g., ($v1$ *relop* $v2$) where $v1$ and $v2$ are primitive program variables (i.e., char, int, float, double) and *relop* is a relational operator], the outcomes of the predicate computation are based on how $v1$ relates to $v2$. For example, Figure 1(b) shows a predicate ($a < m$) at node 3. If $a = 2$ and $m = 1$ when node 3 is executed, for that execution $a > m$ and the predicate outcome is $>$. Our technique therefore assigns $<$, $>$, $==$, and $\perp$ as the set of states for such nodes.

2. If the variables involved in the predicate are pointers or references, our technique introduces states that model pointer or reference equality and inequality, and thus, assigns the states $==$, $\neq$, and $\perp$ to the node.

Modeling Data Flows

Our characterization of the states of certain nodes in terms of data dependences is based on a data flow modeling technique proposed by Laski and Korel [17] as a guide to program testing. Laski and Korel define the *data environment* of a statement s as the set of variable definitions that reach s, along any paths, and are used at s. To more precisely model potential dynamic data flows, Laski and Korel introduced the concepts of elementary data context and data context for a statement s. An *elementary data context* of a statement s, is the set of definitions that reach and are used at a given occurrence of s along some path. The set of all elementary data contexts of a statement s is called the *data context* of the statement.

To illustrate, consider Figure 1(b), and suppose $d_i(x)$ denotes a definition of a variable x at node i. For node 3, the data environment is $\{d_1(m), d_2(a), d_4(a)\}$, the elementary data contexts are $\{d_1(m), d_2(a)\}$ and $\{d_1(m), d_4(a)\}$, and the data context, which is the set of its elementary data contexts, is $\{\{d_1(m), d_2(a)\}, \{d_1(m), d_4(a)\}\}$. For node 4, the data environment and data context are $\{d_2(a), d_4(a)\}$ and $\{\{d_2(a)\}, \{d_4(a)\}\}$, respectively.

The set of states for a PPDG node having data dependence states or substates corresponds to the data context of that node, augmented with the $\perp$ state. The data context of a node characterizes the possible data dependence components of a node's state. Therefore, for example, the states of node 4 are $d_2(a)$,$d_4(a)$, and $\perp$.

If the data context of a node n is the empty set, then by default our technique assigns $\{\top\}$ as its data context. Hence, the states of node n are $\top$ and $\perp$. The state $\top$ means that during a given execution the node was executed. (Recall that $\perp$ means that the node was not executed in a given execution.) For example, the set of states of node 1 in Figure 1(b) is $\{\top, \perp\}$.

Nodes	States	CPDs
1, 2, L4	$\top$, $\perp$	P(1), P(2), P(L4)
D3	($d_1(m)$, $d_2(a)$),($d_1(m)$, $d_4(a)$), $\perp$	P(D3 | 1, 2, 3, 4)
3	$<$, $>$, $==$, $\perp$	P(3 | D3)
4	($d_2(a)$),($d_4(a)$), $\perp$	P(4 | 2, 3, L4)
6	($d_1(m)$, $d_2(a)$),($d_1(m)$, $d_4(a)$), $\perp$	P(6 | 1, 2, 4)

Table 1: Nodes with corresponding states and conditional probability distributions for Prog.

3.1.2 Adding Nodes and Edges to the PDG

The technique adds nodes and edges to the PDG in two cases: (1) if a node has two components or substates (i.e., a predicate substate and a data dependence substate) or (2) if there are self-loops (i.e., nodes that are control or data dependent on themselves) in the PDG.

Nodes with two substates

Predicate nodes often have states based on both predicate outcomes and data dependences. For example, in Figure 1(b), there is a predicate at node 3 but the data environment is $\{d_1(m), d_2(a), d_4(a)\}$. Node 3 has both predicate substates ($\{<, >, ==, \perp\}$) and data dependence substates ($\{(d_1(m), d_2(a)), (d_1(m), d_4(a)), \perp\}$). Our technique transforms such a node into two nodes, each with substates of one kind. The technique makes the node with the data dependence substates the immediate successor of the predicate node's immediate predecessors, and makes the node with the predicate substates an immediate successor of the node with the data dependence substates. The technique also makes the immediate successors of the predicate node, the immediate successors of the node with the predicate substates. Hence, for node 3 our technique introduces node $D3$.

Loops in PDG

A *self-loop* is a directed edge from a node to itself. Loops in a program may cause the program's PDG to contain self-loops. However, self-loops are not permitted in the dependency network formalism on which PPDGs are based. Therefore, our technique eliminates self-loops from a PDG by introducing new nodes and edges. A self-loop in a PDG may involve either a control dependence or a data dependence. If a node is control dependent on itself and the data environment of the node is empty, our technique removes the self-loop and adds a new node to the PDG. If a node is data dependent on itself, our technique removes the self-loop and again adds a new node. Our technique makes the node that was control or data dependent on itself an immediate successor of the new node. The edge from the new node to the immediate successor node is either a control or a data dependence edge depending on the type of self-loop. Our technique assigns the states $\top$ and $\perp$ to the new node.

For example, in Figure 1(b), node 4 is data dependent on itself. Thus, our technique adds node $L4$ to the PDG with a directed edge from $L4$ to node 4. The edge has the same data flow variable as the data flow variable on the self-loop edge. Note that even though node 3 has a self-loop (i.e., it is control dependent on itself), its data environment is not empty. Hence, our technique does not add a new node into the PDG. Instead, our technique treats the node as having two substates. Table 1 shows all the nodes for the example program (`Prog`) with their respective states.

3.2 Learning

Our technique estimates the parameters of the PPDG from the set of execution data ($D = \{D_i\}_{i=0}^{n}$) generated by executing the instrumented program P' with its test suite T_P. Each $D_i \in D$ corresponds to a test case in T_P. Different kinds of execution data (e.g., coverage or trace information) might be used to estimate the parameters of the PPDG—in this paper, our technique uses node-state traces. A *node-state trace* is a sequence of executed nodes, along with their active states, in the transformed PDG. Our technique uses node-state traces to estimate the parameters of the PPDG so that the PPDG will capture some of the temporal behaviors of the program. Each $D_i \in D$ is a node-state trace. A node can appear multiple times in the trace, although the states the node assumes can be different.

In this paper we present a batch-learning algorithm, (shown in Figure 4). However, the algorithm can be modified easily into an on-line learning algorithm.

3.2.1 Estimating Parameters of the PPDG

Learning the parameters of the PPDG consists of estimating conditional probability distributions, which are represented as tables, called *conditional probability tables*, because the states of the nodes in the transformed PDG are discrete. Suppose $X = \{X_1, \cdots, X_n\}$ denotes the set of nodes in the transformed PDG. We denote the ith state associated with node X_j by x_{ji}, the *parents* (immediate predecessors) of a node X_j by $Pa(X_j)$, and the ith assignment of states to the parents of X_j by pa_{ji}.

For a node with no parents, our technique estimates the probabilities ($p(X_j = x_{ji})$) of the nodes as

$$p(X_j = x_{ji}) = \frac{n(X_j = x_{ji})}{n(X_j)} \tag{1}$$

where $n(X_j = x_{ji})$ is the number of times node (X_j) is in state x_{ji} across all node-state traces and $n(X_j)$ is the number of times the node X_j occurs across all node-state traces.

For a node with parents, our technique estimates the probabilities ($p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$) of the node as

$$p(X_j = x_{ji}|Pa(X_j) = pa_{ji}) = \frac{n(X_j = x_{ji}, Pa(X_j) = pa_{ji})}{n(Pa(X_j) = pa_{ji})} \tag{2}$$

where $n(X_j = x_{ji}, Pa(X_j) = pa_{ji})$ is the number of times node X_j and its parents assume a specific state configuration across all node-state traces, and $n(Pa(X_j) = pa_{ji})$ is the number of times $Pa(X_j) = pa_{ji}$ across all node-state traces. A *state configuration* is a set of states assigned to a set of nodes in the PPDG.

Figure 4 shows algorithm, `LearnParam`, that estimates the parameters of a PPDG. The algorithm takes as input a set of execution data D, generated by executing an instrumented program P' with its test suite T_P, and the program's transformed PDG. T he algorithm outputs the PPDG of the program. `LearnParam` traverses each $D_i \in D$ from beginning to end, updating the parent states of nodes and the neccessary counts depending on whether a node in a trace has parents (lines 1 to 9). After `LearnParam` processes D, it computes the conditional probabilities of each node in the transformed PDG (line 10). Finally, it returns the PPDG (line 11). Table 1 shows the conditional probabilities (CPDs) of each node in the transformed PDG of the example program (`Prog`).

Algorithm: LearnParam
Input: $D = \{D_i\}_{i=1}^{n}$; transformed PDG
Output: PPDG
1 **foreach** $D_i \in D$ **do**
2   **for** $j = 1$ *to Length(D_i)* **do**
3   **if** $Pa(X_j) = \emptyset$ **then**
4   increase $n(X_j = x_{ji})$ by 1
5   **else**
6   increase $n(X_j = x_{ji}, Pa(X_j) = pa_{ji})$ by 1
7   **end**
8   **end**
9 **end**
10 compute probabilities of X_j using equations (1), (2)
return PPDG

Figure 4: Batch learning algorithm: LearnParam

3.2.2 Learning Example

For simplicity, we present the estimation of the conditional probability distribution of node 3 in the example program (`Prog`). Note that node 3 is dependent on node D3 in the transformed PDG. Suppose that variables a and m in `Prog` receive the values 3 and 4, respectively. The node-state[3] trace generated by `Prog` is $\{$(1:⊤), (2:⊤), (D3:($d_1(m)$,$d_2(a)$)), (3:<), (4:$d_2(a)$), (L4:⊤), (D3:($d_1(m)$, $d_4(a)$)), (3:==), 6:($d_1(m)$,$d_4(a)$)$\}$. Given this trace, `LearnParam` processes the trace from the beginning, updating the parent states of node 3, until it finds an occurrence of node 3 with its corresponding state in the trace. For this trace, the first occurrence of node 3 has the state < and the current state of node 3's parent is ($d_1(m)$,$d_2(a)$). Therefore, the algorithm increases n(3=<, D3=($d_1(m)$,$d_2(a)$)) by 1. `LearnParam` continues processing the trace until the second occurrence of node 3 is found. The state of node 3 is == and the current state of its parent is ($d_1(m)$, $d_4(a)$). Therefore, `LearnParam` increases n(3=(==), D3=($d_1(m)$,$d_4(a)$)) by 1. At the end of the trace, `LearnParam` computes the probabilities, $p(3 =< |D3 =(d_1(m),d_2(a)))$ and $p(3 = (==)|D3 =(d_1(m),d_4(a)))$, which are 1.0 and 1.0, respectively. All other entries in the conditional probability table will be 0.

4. APPLICATIONS OF THE PPDG

In this section, we present two applications of the PPDG to software engineering tasks. For the first task, fault localization, we show how the PPDG can be used to overcome some of the limitations of current fault-localization techniques, and introduce a simple ranking-based algorithm that analyzes a faulty execution using the PPDG to determine the most suspicious statements in the program. For the second task, fault comprehension, we also exploit the interpretive nature of the PPDG, and present an algorithm that generates contextual information related to suspicious statements—information that indicates why a particular statement is considered highly suspicious.

4.1 Fault Localization

Debugging software is often a difficult and time-consuming task, which must be done manually. One of the most laborious aspects of debugging is *fault localization*—locating the fault or faults in a program that caused one or more observed failures. To reduce the burden on the developer during fault

[3] We denote each node-state in the trace as "(node:state)".

Algorithm: RankCP

Input: node-state trace:$\{X_j : x_{ji}\}_{j=1}^{n}$; PPDG
Output: ranked nodes with state configurations

```
1  for j = 1 to n do
2      prob ← p(X_j = x_ji | Pa(X_j) = pa_ji)
3      if prob < lowest_prob(X_j) then
4          lowest_prob(X_j) ← prob
5          index(X_j) ← j
6          configuration(X_j) ← {x_ji ∪ pa_ji}
7      end
8  end
9  rank nodes in ascending order by probability, break ties
   using indexes
10 return ranked nodes with state configurations
```

Figure 5: RankCP algorithm.

localization, a number of fault-localization techniques (e.g., [5, 14, 16, 18, 19, 25, 28, 29]) have been developed.

Existing fault-localization techniques fall into two main categories: those that require knowledge of the incorrectness of the values of program variables and those that do not require this knowledge. Techniques that require knowledge of incorrectness of variable values are mostly slicing techniques [28, 29]. The limitation of the slicing techniques is that they do not provide a ranking of the statements in the slices presented to the developer. This lack of guidance as to how the statements in a slice should be examined may increase the burden of finding the faulty statement.

The techniques that do not require knowledge of the incorrectness of the values of program variables can be divided into two main groups. The first group [14, 16, 18, 19] requires access to multiple executions that fail because of the fault, as well as access to multiple passing executions. The second group [5, 25] requires access to multiple passing executions and access to only one execution that fails because of the fault. The first group of techniques has been shown through published results (i.e., [15, 19]) to be more effective in localizing faults than the second group. However, in practice it is not always possible to have access to multiple executions that fail because of a given fault.

Figure 5 shows algorithm `RankCP`, which analyzes a single failing execution at a time, and ranks[4] nodes in the PPDG. `RankCP` ranks nodes based on the conditional probabilities of nodes given their parent nodes (i.e., $p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$). `RankCP` uses the conditional probabilities of nodes given their parent nodes because the conditional probabilities measure how nodes are influenced by their parent nodes in the PPDG. `RankCP` ranks the nodes with the lowest probability as highly suspicious. The conditional probability, $p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$, chosen as an inverse measure of suspiciousness, is based on preliminary studies we conducted that showed faults tend to be associated with low probability nodes.

`RankCP` inputs, for a program, a node-state trace generated by a failing execution and the PPDG. `RankCP` returns a list of nodes ranked from most suspicious to least suspicious, where suspiciousness is inversely proportional to the lowest conditional probability of any of the node's states. (Recall that a node can occur multiple times in a trace.) Each node is also associated with a node-parent state configuration. `RankCP` processes a trace from beginning to end. As it processes the trace (line 1), it computes the conditional probability of a node's current state given the current states of its parents (i.e., $p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$) (line 2). Then, `RankCP` records for each node, the lowest value *lowest_prob* of this probability (lines 3 and 4). Note that `RankCP` computes $p(X_j = x_{ji}|Pa(X_j) = pa_{ji})$ using the probability in the conditional probability table for node X_j. `RankCP` also keeps track of the index of nodes in the trace in the *index* variable (line 5). `RankCP` associates with each node a node-parent state configuration using the *configuration* variable (line 6). After `RankCP` has processed the trace, it ranks the nodes by their *lowest_prob* values, and if two nodes have the same *lowest_prob* values, the algorithm ranks the node with the lower *index* value higher (line 9). The algorithm returns the ranked nodes with their associated state configurations (line 10). The ranked nodes with their associated state configurations are retained for use in fault comprehension.

Sections 5.2 and 5.3 present studies that illustrate the potential effectiveness of using the PPDG for fault localization.

4.2 Fault Comprehension

Another important aspect of debugging is *fault comprehension*—understanding the reason why a faulty statement or set of statements causes failures. However, the fault comprehension problem has not received as much attention from researchers as the fault-localization problem. Jiang and Su [14] present a technique that constructs faulty control flow paths from program predicates, which are intended to aid fault comprehension. Cleve and Zeller [5] introduced a technique that provides contextual information, in the form of cause-effect chains, that is useful for fault comprehension.

The fault-comprehension algorithm, `FaultComp`, generates contextual information (i.e., explanations) from the PPDG by analyzing the state configurations involving a node and its parents. The contextual information is based on the state configuration of a node and its parents because `RankCP` ranks nodes based on the conditional probability of a node given its parent nodes. The state configurations generated from the PPDG that relates a node and its parents are called expected state configurations. An *expected state configuration* is a state configuration generated from the PPDG whose probability is greater than zero. The probabilities of the expected state configurations indicate the likelihood of the configurations. Note that a node may have several expected state configurations.

Figure 6 shows algorithm `FaultComp`, which generates explanations from the PPDG. `FaultComp` inputs, for a program, a set of ranked nodes with their node-parent state configurations obtained from the `RankCP` algorithm. `FaultComp` also inputs the PPDG of the program. `FaultComp` outputs a set of nodes, with each node having an expected configuration set. `FaultComp` processes the set of ranked nodes (line 1) and, for each node X_j, it extracts the state of X_j (i.e., x_{ji}) from X_j's node-parent state configuration C_j (line 2). `FaultComp` uses this state to extract the state configurations for X_j's parents from the PPDG (line 3)—`FaultComp` extracts all the conditional probabilities, $p(X_j = x_{ji}|Pa(X_j))$, that are greater than zero. Thus, only parent state config-

[4]Note that, `RankCP` can rank nodes in the PPDG using different probability measures (e.g., marginal probabilities, conditional probabilities, and joint probabilities). Studying the effects and trade-offs of different probability measures is part of our current research but not presented in this paper.

Algorithm: FaultComp

Input: ranked nodes with faulty state configurations:$\{X_j : C_j\}_{j=0}^k$; PPDG

Output: nodes with ranked expected-configuration sets

```
1 foreach X_j ∈ Ranked nodes do
2     get state of X_j from C_j
3     get parent state-configurations associated with state
      of X_j with probabilities greater than zero from X_j's
      conditional probability table
4     add parent state-configurations of X_j with
      probabilities to expected configuration set
5 end
6 rank state-configurations in the expected configuration
  set of each node from highest probability to lowest
7 return nodes with ranked expected-configuration sets
```

Figure 6: FaultComp algorithm.

urations with probabilities greater than zero are extracted. `FaultComp` stores the parent state configurations and their probabilities in an expected configuration set (line 4). After processing all the nodes, `FaultComp` ranks the configurations in decreasing order of their probabilities (line 6), and returns the ranked expected configurations (line 7). The parent state configuration with the highest probability offers the best explanation of what the expected behavior of the node and its parents should be.

Section 5.4 presents a case study that shows the potential utility of using PPDGs to facilitate understanding of faults.

4.3 Example

To illustrate the application of the PPDG, we provide an example of how `RankCP` and `FaultComp` are used to localize and generate contextual information related to faults in a given failing execution. Suppose that variables a and m in `Prog` in Figure 1 receive the values 3 and 4, respectively, and that `Prog` behaves correctly under this input. The node-state trace generated is $\{(1{:}\top)$, $(2{:}\top)$, $(D3{:}(d_1(m),d_2(a)))$, $(3{:}<)$, $(4{:}d_2(a))$, $(L4{:}\top)$, $(D3{:}(d_1(m), d_4(a)))$, $(3{:}{=}{=})$, $(6{:}(d_1(m),d_4(a))\}$. Suppose further that the parameters of the PPDG are estimated using this node-state trace. Now, suppose that a and m receive the inputs 5 and 1, respectively, and this input causes `Prog` to fail. The failing node-state trace generated is $\{(1{:}\top)$, $(2{:}\top)$, $(D3{:}(d_1(m),d_2(a)))$, $(3{:}>)$, $(6{:}(d_1(m),d_2(a))\}$. Given this failing execution, `RankCP` will flag nodes 3 and 6 as suspicious because, according to the PPDG, $p(3 => |D3 = (d_1(m), d_2(a))) = 0.0$ and $p(6 = (d_1(m), d_2(a))|1 = \top, 2 = \top, 4 = \bot) = 0.0$, respectively. Because nodes 3 and 6 have the same probability, `RankCP` will use the index of the nodes in the failing node-state trace to order the ranking. Node 3 is ranked higher than node 6 because the index of nodes 3 and 6 are 4 and 5, respectively. Suppose node 3 is faulty, then `RankCP` will associate the state configuration $(3 :> \cup (D3 = (d_1(m), d_2(a)))$ with node 3. `FaultComp` will use the state of node 3, $>$, to generate the expected state configurations. However, there are no state configurations associated with the state of node 3 (i.e., $>$) in the PPDG. Hence, `FaultComp` does not associate any expected state configuration with node 3, which implies that the parent of node 3 (i.e., $D3$) never caused node 3 to get into the state $>$ during learning. Therefore, that is the reason `RankCP` ranks node 3 higher.

5. EMPIRICAL EVALUATION

To evaluate the effectiveness of the `RankCP` algorithm when applied to the fault-localization problem, we compared it to existing fault-localization techniques: `RankCP` to `SOBER` [19], `Tarantula` [16], and Cause Transitions (`CT`) [5]. We performed both experiments and case studies on fault localization. We also performed a case study on fault comprehension.

5.1 Experiment Setup

We used the *Siemens suite* [13] as our subjects in our studies; it is the most common set of subjects used to determine the effectiveness of fault-localization techniques. Table 2 shows the characteristics of the seven Siemens programs: the name of the program, the number of faulty versions, the number of lines of code, the number of nodes in PPDG, the number of test cases, and a description of the program. There are 132 faulty versions in total and each program is associated with a matrix that indicates which test cases pass and which test cases fail. Each faulty version has exactly one fault. For our experiments, we omitted eight versions: versions 8, 14, and 32 of *replace*, versions 4 and 6 of *print_tokens2*, version 9 of *schedule*, version 9 of *schedule2*, and version 38 of *tcas*. We omitted these versions because (1) there were no syntactic differences between the C file of the correct version of the program and the faulty version (e.g., change in header file), (2) no traces could be gathered because the faulty versions had segmentation faults when executed on their test suite, or (3) none of the test cases failed when executed on the faulty version of the program. The other experiments also eliminated some versions: `SOBER` used a total of 130 versions, `Tarantula` used 122 versions, and `CT` used 129 versions.

We implemented the algorithm to build the PPDG in the `Objective Caml` language. Our technique uses the CIL framework [22] to analyze and instrument the source files of C programs. CIL transforms all conditions with compound predicates (i.e., conjunctions and/or disjunctions of simple predicates) into conditions with simple predicates.[5] Our technique automatically performs the predicate transformations discussed in Section 3.1.1 and builds the PDG for each faulty version.

5.2 Fault Localization

For fault localization, our technique builds a PPDG for each faulty version of the program. Our technique used the traces of passing test cases to estimate the parameters of the PPDG. Using passing test cases to estimate the parameters of the PPDG enables the PPDG to capture the correct behaviors of the program exposed by passing test cases. After building the PPDG, we ran the `RankCP` algorithm on the trace of each failing test case. (Recall that `RankCP` analyzes a single trace at a time.)

5.2.1 Study 1: Effectiveness

The goal of this study is to compare `RankCP` to other fault localization techniques in terms of effectiveness. We obtained the fault localization results for `Tarantula`, `CT`, and `SOBER` from published results [5, 15, 19].

[5]Note that if a predicate in a condition consists of a function call, CIL evaluates the function separately and stores the return value in a temporary variable, which is then used in the predicate.

Program	Faulty Versions	LOC	PPDG Size	Test cases	Description
print_tokens	7	472	930	4130	lexical analyzer
print_tokens2	10	399	290	4115	lexical analyzer
replace	32	512	397	5542	pattern replacement
schedule	9	292	201	2710	priority scheduler
schedule2	10	301	212	2650	priority scheduler
tcas	41	141	130	1608	altitude separation
tot-info	23	440	252	1052	information measure

Table 2: Subjects used for experiments.

To evaluate the effectiveness of RankCP to the other fault-localization techniques, we use the metric, *score*, which was used by References [5, 15, 25]. *Score* represents the percentage of nodes that must be examined by the developer to find the fault, assuming the developer starts from the highest ranked suspicious node and examines nodes in decreasing order of suspiciousness until the faulty node is found. For example, a *score* range of $0\% - 1\%$ implies the developer needs to examine less than 1% of the code to find the fault. The *score* is computed as

$$\text{Score} = \frac{|\text{N}|}{|\text{PPDG}|} \times 100 \qquad (3)$$

where |N| is the number of nodes examined to find faulty node and |PPDG| is the number of nodes in the PPDG.

Because RankCP analyzes a single failing trace at a time, we show its best case and worst case performances on the set of failing test cases for each faulty version. For example, for version 31 of *replace*, there are 210 failing test cases. For 95.3% (i.e., approximately 200) of the failing test cases, less than 1% of the code must be examined to find the faulty statement. For the remaining 4.7% (i.e., approximately 10) of the failing test cases, between 1% and 10% (exclusive) of the code must be examined. This implies the best *score* range for RankCP for the faulty version is $0\% - 1\%$ and the worst is $1\% - 10\%$.

Table 3 shows the percentage of faults found at each *score* range for each of the techniques. RankCP-best and RankCP-worst represents the best and worst performance of RankCP, respectively. Under RankCP-best, for 41.94% of the faulty versions, less than 1% of the program must be examined to locate the faulty statement. Under RankCP-worst, for 17.74% of the faulty versions, less than 1% of the program must be examined to find the faulty statement. When less than 1% of the code must be examined under RankCP-best, our technique is approximately 9, 5, and 3 times more effective than CT, SOBER, and Tarantula, respectively. Under RankCP-worst, our technique is approximately 4 and 2 times more effective than CT and SOBER, respectively.

Figure 7 shows the cumulative results of Table 3. The horizontal axis represents the percentage of a program's statements that must be examined to find the fault it contains and the vertical axis represents the percentage of faulty versions that are found given a *score* on the horizontal axis. Note that the vertical axis can also be interpreted as the percentage of faults found if no more than a given percentage of the program is examined. The legend lists the fault-localization techniques. Figure 7 shows that RankCP-best and RankCP-worst are more effective than CT. This is significant because RankCP and CT both analyze a single failing execution at a time.

Score	RankCP-best	RankCP-worst	Tarantula	CT	SOBER
0-1%	41.94	17.74	13.93	4.65	8.46
1-10%	31.45	27.42	41.80	21.71	43.84
10-20%	13.71	25.81	5.74	11.63	21.54
20-30%	2.42	4.84	9.84	13.18	3.85
30-40%	2.42	4.84	8.20	1.55	4.62
40-50%	5.65	8.06	7.38	6.98	0.77
50-60%	1.61	2.42	0.82	3.10	0.77
60-70%	0.0	5.65	0.82	7.75	2.31
70-80%	0.8	2.42	4.10	4.65	2.31
80-90%	0.0	0.81	7.38	6.98	2.31
90-100%	0.0	0.0	0.00	17.83	9.23

Table 3: Percentage of located faults w.r.t percentage of code examined.

5.2.2 Study 2: Efficiency

The goal of this study is to determine the efficiency of our technique and to compare it to the efficiency of other fault-localization techniques. We conducted our efficiency experiments on a 3.2 GHz Intel Pentium-4 PC with 2 GB of memory. We obtained timings for Tarantula and CT from published results [5, 15].

Table 4 summarizes the results of the study. The columns show the programs, the average time taken to process all traces and build the PPDG, the average computation time taken by RankCP to analyze a single failing execution, the computation time of Tarantula, and the average computation time of CT, respectively. All the timings are in seconds. As the results show, the time required to process all traces and build the PPDG, in addition to the computation time required by RankCP to localize the fault in a given failing execution, is less than the computation time of CT. For example, for *replace*, our technique requires, on average, less than 6 minutes to process all traces and build the PPDG. The computation time for CT for *replace* is approximately 1 hour. However, the computation time for RankCP is, on average, 0.0327 seconds. The timings are signficant because both CT and RankCP analyze a single failing execution at a time. Of all the techniques, Tarantula is the most efficient; Tarantula takes milliseconds to finish its fault-localization analysis. Note that none of our implementations have been optimized. Furthermore, differences in computing environments (e.g., operating systems and programming languages) might affect the results. Therefore, the efficiency results should not be viewed as definitive.

5.3 Fault Localization: Case Study

The goal of this study is to further determine the effectiveness and scalability of our technique. To do this, we performed a case study using the *Sed* program. *Sed* is a stream editing utility for the Unix Operating System platform that

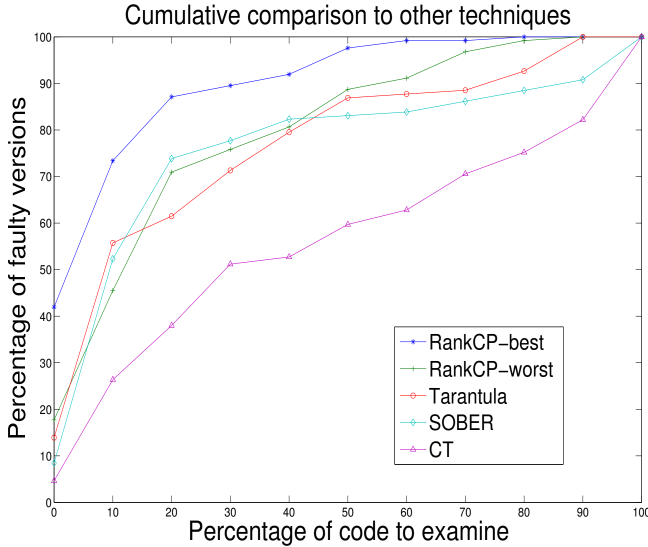


Figure 7: Cumulative comparison with other techniques on the Siemens subjects

Program	PPDG (Process traces & build)	RankCP (Computation time)	Tarantula (Computation time)	CT (Computation time)
print_tokens	846.5	0.2176	0.0040	2590.1
print_tokens2	243.6	0.0574	0.0037	6556.5
replace	335.3	0.0327	0.0063	3588.9
schedule	77.3	0.0082	0.0032	1909.3
schedule2	199.5	0.0217	0.0030	7741.2
tcas	1.7	0.0003	0.0025	184.8
tot_info	97.6	0.0605	0.0031	521.4

Table 4: Efficiency of technique in seconds.

contains 14K lines of code. We obtained the software from the Software-artifact Infrastructure Repository [6]. *Sed* has seven versions, each with a number of seeded faults that can be activated individually. For our study, we randomly chose versions 4, 5, and 6. We activated all faults in the versions individually, which resulted in 14 faulty versions of *Sed*. Each faulty version had a single fault. Out of the 14 faulty versions, we omitted three versions because none of the test cases failed on them. The number of test cases is between 360 and 370 for each of the versions. The subject also comes with a matrix that indicates the test cases that pass and the test cases that fail.

Table 5 shows the results of the study. The faulty-version column shows which version of the *Sed* program was used and which fault was activated. For example, V6-F1 means version 6 of *Sed* was used with the first fault activated. MBT (model building time) gives the time it took to build the PPDG, `RankCP`-best, `RankCP`-worst, and `RankCP`-median give the best, worst, and median fault-localization results, respectively, for the faulty versions. For *Sed*, to measure the effectiveness of `RankCP`, we use the number of nodes in the PPDG that must be examined to find the faulty statement instead of the percentage of the program that must be examined. Using the number-of-nodes metric gives us a better view of the effectiveness of `RankCP`.

As Table 5 shows, our technique was effective at localizing the faults in some faulty versions of *Sed* that we examined.

Faulty Version	MBT (seconds)	PPDG Size	RankCP-best	RankCP-worst	RankCP-median
V4 - F2	1779.91	7049	2	2	2
V5 - F1	713.40	9137	10	429	20
V5 - F2	723.02	9138	2	15	4
V5 - F3	745.71	9137	264	1370	429
V5 - F4	750.13	9138	317	318	317
V6 - F1	614.14	9142	1	143	4
V6 - F2	330.81	9138	7	2941	2015
V6 - F3	334.24	9143	7	2940	2015
V6 - F4	735.62	9142	7	7	7
V6 - F5	569.43	9142	1983	2702	2237
V6 - F6	798.47	9137	2093	2483	2247

Table 5: Fault localization case study.

For example, for V5-F2 under `RankCP`-best, only the top two nodes must be examined to find the faulty node, and under `RankCP`-worst and `RankCP`-median, only 15 and 4 nodes must be examined, respectively, to find the fault. However, for some versions `RankCP` was not effective—a large number of nodes must be examined. For example, for V6-F6, the developer must examine 2093 and 2483 nodes under `RankCP`-best and `RankCP`-worst, respectively.

Table 5 also shows that our technique can be made to scale to large programs—the technique required less than an hour to build the PPDG for each faulty version of *Sed*.

5.4 Fault Comprehension: Case Study

The goal of this study is to determine the potential usefulness of PPDGs for fault comprehension. To do this, we selected V6-F5 of *Sed*. Figure 8 shows a graph of the explanation generated by `FaultComp` from the PPDG, relating nodes 2964 and 2965 to their parents. The labels on the nodes correspond to the line numbers of the statements in V6-F5 of *Sed*. The fault in V6-F5, which is an off-by-one error, is located at node 2950. The result of the computation that occurs at node 2950 is used at nodes 2964 and 2965. Table 5 shows that for V6-F5, `RankCP`-best and `RankCP`-worst will require the developer to examine 1983 and 2702 nodes, repectively, to find the fault. However, `RankCP` ranks nodes 2964 and 2965 as the first and second suspicious nodes, respectively, but if the developer examines the explanation generated by `FaultComp` for node 2964, only nodes 2964, 2950, and 2833 must be examined to find the fault, The use of the explanation significantly reduces the time required to find the fault.

On closer examination of the explanation generated by `FaultComp` for node 2964, we would see that the parent states of node 2964 never caused the node to be in state $<$. Also on closer examination of the explanation generated for node 2965, the node gets into an unknown state (i.e., the state was never encountered during building of the PPDG). The unknown state occurs because the result of the computation at node 2950 is also used at node 2965, which is control dependent on node 2964. These explanations generated by `FaultComp` provide reasons why nodes 2964 and 2965 are ranked higher by `RankCP`.

6. RELATED WORK

There are a number of techniques, that have used probabilistic graphical models to address software engineering tasks. Burnell and Horvitz [4] use belief networks (Bayesian networks [21]) for debugging of mainframe assembler programs. Their technique is similar to ours in that they gen-

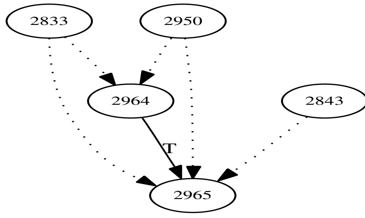


Figure 8: Graph of explanation generated by `FaultComp` from PPDG.

erate explanations from the belief networks. However, the main difference between their belief network and the PPDG is that their belief network does not capture the statistical dependences among the elements in a program. Also their technique focuses on mainframe assembler programs but PPDGs is applicable to any program whose PDG can be obtained.

There are a number of techniques (e.g., [3, 10, 24]) that build models of program behaviors. Bowring and colleagues [3] build models of program behaviors using Markov models. The models are constructed from program entities such as branches and method calls. Haran and colleagues [10] build models of program behaviors using tree-based classifiers. Podgurski and colleagues [24] build models of program behaviors using automated clustering techniques. The classifiers built from their models are later used to classify software executions. The PPDG differs from their models because it models statistical dependencies between program elements, which potentially makes the PPDG more accurate because of the semantic information obtained by capturing statistical dependencies between program elements.

There are also temporal specification-mining techniques (e.g., [1, 2, 27]) that extract models from programs. These techniques mine specifications from programs and use them to detect errors in programs. The kind of errors the techniques can detect are the errors that violate the specifications. The PPDG is similar to their models in that it encodes a probabilistic specification of a program thus making PPDGs useful for fault diagnosis. The difference is that the specification miners are limited in the kind of errors they can detect. The PPDG, however, has the potential to detect a wide variety of errors because it captures the statistical dependencies between program elements.

7. CONCLUSION AND FUTURE WORK

In this paper, we present the PPDG, a probabilistic graphical model based on the PDG that captures the statistical dependences among program elements and enables the use of probabilistic reasoning to analyze program behaviors. We also presented algorithms for two applications of the PPDG: `RankCP`, which uses the PPDG to rank statements to assist in fault localization and `FaultComp`, which uses the PPDG to generate explanations to aid in fault comprehension. To evaluate the PPDG, we implemented its construction, `RankCP`, and `FaultComp`, and performed several studies. The results of the studies show the potential usefulness of the PPDG for these two software engineering tasks.

Our studies show that, in many cases, `RankCP` is effective for fault localization. However, the algorithm is not effective in localizing faults in some failing executions—in these cases, a large number of statements must be examined to find the fault. One reason for this ineffectiveness is that `RankCP` ranks nodes in the PPDG using the conditional probabilities of nodes and their parents. Thus, the algorithm may not localize faults whose effects transcend node-parent state configurations. We are currently working on new algorithms that consider local and global effects of faults.

Our studies also show that the PPDG can be an effective model for representing the behaviors of a program for fault diagnosis. However, the efficacy of the PPDG depends on the abstract states used by nodes to capture the semantics of computations. For example, for memory-related faults, `RankCP's` was not effective during fault localization because the states of nodes in the PPDG did not capture memory related behaviors. Because the state information directly affects the efficacy of the PPDG, we are investigating other types of states (e.g., object states and predicate abstractions) to identify those that might better represent the semantics of the computations.

One critical part of our PPDG construction is the execution information, which is used to estimate the paramters of the PPDG. This execution information is dependent on the test suite that is executed by the instrumented program. In our experiments, we used large test suites that had been used for many previous testing and debugging experiments, and thus, provided good coverage of program behaviors. However, in general, we have not determined criteria for selecting appropriate test-suites for used with our technique. We are currently investigating such criteria.

Our efficiency experiment, although limited, suggests that our technique is more efficient than existing techniques that consider single failing executions. However, the overhead imposed by source-code instrumentation limits the application of PPDGs to large software systems and to deployed software. We are currently developing fault-diagnosing algorithms that require only partial information (e.g., coverage data as opposed to traces) that could make our technique applicable to these systems.

The PPDG we used in this paper was based on intraprocedural PDGs, and thus the PPDGs used for our studies do not capture statistical dependences across functions. In the future, we will base the PPDG on the interprocedural PDG, enabling the PPDG to capture the statistical dependences among program elements (e.g., pointers and references) whose behaviors are not confined to a single function.

We have shown the potential utility of applying the PPDG to the problem of fault diagnosis but we believe that it has other applications. We therefore plan to investigate the potential application of the PPDG to other software engineering tasks.

Acknowledgements

This work was supported in part by NSF awards CCF-0429117, CCF-0541049, and CCF-0725202 to Georgia Tech. The anonymous reviewers provided many comments that helped improved the presentation of this paper.

8. REFERENCES

[1] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java classes. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 98–109, January 2005.

[2] G. Ammons, R. Bodik, and J. R. Larus. Mining Specifications. In *Symposium on Principles of Programming Languages*, pages 4–16, January 2002.

[3] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active Learning for Automatic Classification of Software Behavior. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 195–205, July 2004.

[4] L. Burnell and E. Horvitz. Structure and Chance: Melding Logic and Probability for Software Debugging. *Communications of the ACM*, 38(3):31–41., 1995.

[5] H. Cleve and A. Zeller. Locating Causes of Program Failures. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages pages 342–351, May 2005.

[6] H. Do, S. Elbaum, and G. Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Trans. on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] S. Galan, F. Aguado, F.J.Diez, and J. Mira. NasoNet, Joining Bayesian Networks, and Time to Model Nasopharyngeal Cancer Spread. *Artificial Intelligence in Medicine*, 2101/2001:207–216, 2001.

[9] K. Gallagher. *Using Program Slicing in Software Maintenance*. PhD thesis, University of Maryland, 1989.

[10] M. Haran, A. Karr, A. Orso, A. Porter, and A. Sanil. Applying Classification Techniques to Remotely-Collected Program Execution Data. In *Proceedings of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005)*, pages 146–155, September 2005.

[11] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, and S. Spoon. Regression Test Selection for Java Software. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2001)*, pages 312–326, October 2001.

[12] D. Heckerman, D. M. Chickering, C. Meek, R. Rounthwaite, and C. M. Kadie. Dependency Networks for Inference, Collaborative Filtering, and Data Visualization. *Journal of Machine Learning Research*, 1:49–75, 2000.

[13] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the Effectiveness of Dataflow and Controlflow-Based Test Adequacy Criteria. In *International Conference on Software Engineering*, pages 191–200, May 1994.

[14] L. Jiang and Z. Su. Context-Aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, November 2007.

[15] J. Jones and M. J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, November 2005.

[16] J. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, pages 467–477, May 2002.

[17] J. W. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Transactions on Software Engineering*, 9:347–354, 1983.

[18] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 15–26, June 2005.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER:Statistical Model-based Bug Localization. In *Proceedings of the 5th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 286–295, September 2005.

[20] K. Murphy. *Dynamic Bayesian Networks: Representation, Inference and Learning*. PhD thesis, UC Berkeley, Computer Science Division, 2002.

[21] R. E. Neapolitan. *Learning Bayesian Networks*. Prentice Hall, 2003.

[22] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, April 2002.

[23] A. Podgurski and L. A. Clarke. A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[24] A. Podgurski, D. Leon, P. Francis, W. Masri, M. M. Sun, and B. Wang. Automated Support for Classifying Software Failure Reports. In *Proceedings of the 25th International Conference on Software Engineering*, pages 465–475, May 2003.

[25] M. Renieris and S. Reiss. Fault Localization With Nearest Neighbor Queries. In *International Conference on Automated Software Engineering*, pages 30–39, November 2003.

[26] S. Thrun. Robotic Mapping: A Survey. *In Exploring Artificial Intelligence in the New Millennium*, pages 1–35, 2002.

[27] W. Weimer and G. Necula. Mining Temporal Specifications for Error Detection. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 461–476, April 2005.

[28] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, March 1981.

[29] X. Zhang, N. Gupta, and R. Gupta. Pruning Dynamic Slices With Confidence. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 169–180, June 2006.