American Journal of Engineering Research (AJER)
e-ISSN : 2320-0847 p-ISSN : 2320-0936
Volume-03, Issue-05, pp-64-72
www.ajer.org

Research Paper Open Access

An Empirical Study of the Probabilistic Program Dependence Graph

Ms. Pushpanjali Patra[1], Ms. Syeda Ayesha Thainiath[2]
Assistant Professor in Computer Science Engineering at Nawab Shah College Of Engineering & Technology (Affliated to JNTUH), Malekpet, Hyderabad-500024, A.P, India.
Assistant Professor in Computer Science Engineering at Nawab Shah College Of Engineering & Technology (Affliated to JNTUH), Malekpet, Hyderabad-500024, A.P, India.

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), which facilitates probabilistic analysis and reasoning about uncertain program behavior particularly that associated with faults. The PPDG construction augments 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 used widely in a variety of applications? This project presents algorithms for constructing PPDGs and applying them to fault diagnosis. The project also presents preliminary evidence indicating that a PPDG-based fault localization technique compares favorably with existing techniques. The project also presents evidence indicating that PPDGs can be useful for fault comprehension.
The larger, more complex a program, the higher the likelihood of it containing bugs. It is always challenging for programmers to effectively and efficiently remove bugs, while not inadvertently introducing new one sat the same time. Furthermore, to debug, programmers must first be able to identify exactly where the bugs are, which is known as fault localization.

Keywords: *Fault Localization, Probabilistic Program Dependence Graph (PPDG).*

I. INTRODUCTION

The program dependence graph can be used to construct a novel and useful probabilistic graphical model of program behaviour. The model captures the conditional statistical dependence and independence relationships among program elements in a way that facilitates making probabilistic inferences about program behaviours. We call this model a Probabilistic Program Dependence Graph (PPDG).

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. Program dependence graphs (PDGs), which have proven useful in software engineering applications such as testing, debugging, and maintenance between program elements. It augments program dependence graphs with statistical dependence (and independence) information in the principled way provided by probabilistic graphical models, it is possible to substantially increase the utility of program dependence graphs in some software engineering applications.

Probabilistic graphical models have proven useful in several fields (e.g., medicine and robotics) 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 program dependence graph provides a natural framework for modelling both the presence of dependences and their statistical strengths.

Our technique produces the PPDG for a program by augmenting its program dependence graph automatically. 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.

Intuitively, PPDGs are well suited to these tasks for two reasons. First, they can indicate how a failing execution differs from successful ones, both structurally and statistically. Second, context information generated from PPDGs can be used 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 behaviours.

The main contributions of the paper are the following:
- The 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.

Existing System

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. Graphical 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 behaviours are. This severely limits their utility for reasoning about the causes and effects of inherently uncertain program behaviours, such as runtime failures.

Proposed System

We show how the program dependence graph can be used to construct a novel and useful probabilistic graphical model of program behaviour. The model captures the conditional statistical dependence and independence relationships among program elements in a way that facilitates making probabilistic inferences about program behaviours. 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 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.

Proposed System Features

Identify the fault comprehension and fault localization process environment process. The PPDG, a novel probabilistic graphical model of program behaviour based on the program dependence graph, applications of the PPDG to fault localization and fault comprehension, and. the results of empirical studies that show that the PPDG can be useful for these applications.

II. ARCHITECTURE

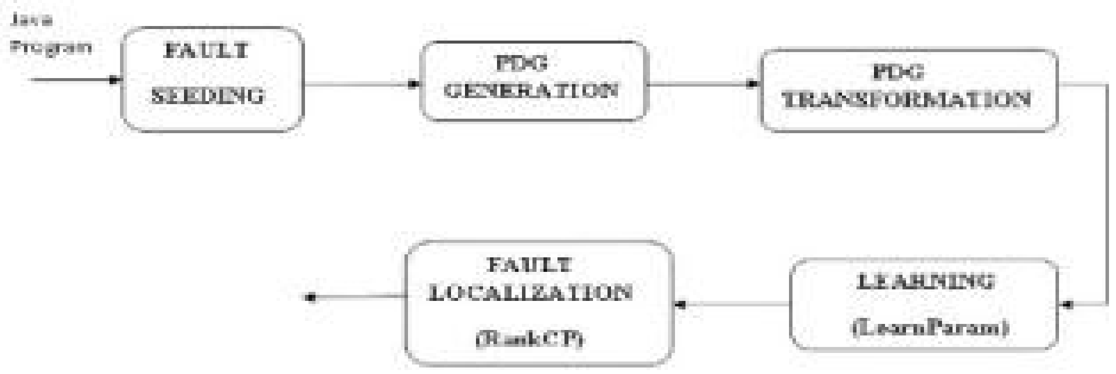


Fig 1- ARCHITECTURE

Our approach involves the PPDG generation and fault localization. PPDG is an innovative model of a program's internal behaviour over a set of test inputs. It facilitates probabilistic analysis and reasoning about uncertain program behaviour, particularly those associated with faults. The PPDG is based on the established framework of probabilistic graphical models. It scans each and every state nodes for fault in a program. Since it is a graphical representation testers can easily find exactly where the fault is. PPDG is nothing but transformed PDG. This transformation is achieved by learning. For learning LearnParam algorithm is used. This will transform the predicate nodes and self loop nodes by adding additional node as its parent.

Learn Aram algorithm is used to generate the PPDG. Fault localization is done by RankCP algorithm. It is used to find the probabilistic distribution of each node. It ranks each node using probability and the node having less probability is considered to be most suspicious. In LearnParam algorithm the execution trace is taken as input and evaluates the probability for each node based on the dependences to the node.

1 PDG transformation

During this step, our technique

1) Structurally transforms the PDG by adding nodes and edges to it

2) Specifies the states of the nodes.

We call the graph that results after transforming the PDG the transformed PDG. 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 exclusive1 (i.e., a node cannot be in two different states at the same time). Our technique initially assigns a default state denoted by the symbol? To each node. The State is the state a node assumes when it has not been executed. When a node is executed, it assumes a state distinct from?

The state of a PPDG node abstracts a part of the program's state that pertains to the node when the program executes. There are different ways to model this "local" concrete state. We model it in one or both of two 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. Our technique characterizes the state of a node representing a branch predicate by the outcome of the predicate.

The technique characterizes the state of a node representing a statement s that uses one or more variables by the set of variable definitions that reaches those uses during execution (i.e., by the definitions on which s is dynamically datadependent2).Our technique transforms all predicates 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 or disjunctions of simple predicates) are transformed into conditions with simple predicates. If a condition (e.g., "ifðv1Þ") consists of a single variable (i.e., v1), our technique treats the condition as"ifðv1 ¼¼ 0Þ". Hence, the predicate for the condition is"v1 ¼¼ 0" (i.e., v2 is 0). Transforming all predicates into simple predicates simplifies the transformation of the PDG.

1.1 Structural transformation

Our technique adds nodes and edges to the PDG in two cases:

1) If a node has two state components (i.e., a predicate component and a data dependence component) or

2) If there are self-loops (i.e., nodes that are control or data dependent on themselves) in the PDG. Transforming nodes with two state components.

The state of a predicate node can be characterized by both a predicate outcome and a set of dynamic data dependences. Thus, the state of a predicate node may have two state components (i.e., a predicate component and a data dependence component). If so, our technique introduces a new node into the PDG and assigns the data dependence component to the new state (removing it from the predicate node). The technique makes the new node the immediate successor of the predicate node's immediate predecessors and makes the original predicate node an immediate successor of the new node. Note that the predicate no deretains its connection to its immediate successors. For example, the predicates "i < n" and "v > max" at nodes 4 and 6 , respectively, each have two state components. Predicate "i < n" has two state components because of the predicate computation at the node and because of its dynamic data dependences on nodes 1, 2, and8. Predicate "v > max" has two state components because of the predicate computation at the node and because of its dynamic data dependences on nodes 3, 5, and shows the result of the structural transformation of the PDG of find max that introduces new nodes D4 and D6.Transforming self-loops. 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 control dependence or data dependence. If a node n is data dependent on itself with respect to a program variable v, our technique removes the self-loop transformation.

If a node is control dependent on itself and the predicate at the node has two state components, our technique does not add a new node to the PDG. Because the predicate node had two state components in the previous step, it was already transformed and a node was added. Our technique moves the self-loop and connects a control dependence edge from the original predicate node to the node added in the previous step.

For example, shows that node 4 is control dependent on itself. However, a new node is not added because the predicate at the node has two state components, and therefore, has already been transformed. Instead, the technique adds a control dependence edge from node 4 to node D4. Shows the result of this transformation. Shows the result of structurally transforming the PDG of example program find max. This graph structure forms the structure of the PPDG (dependency network).

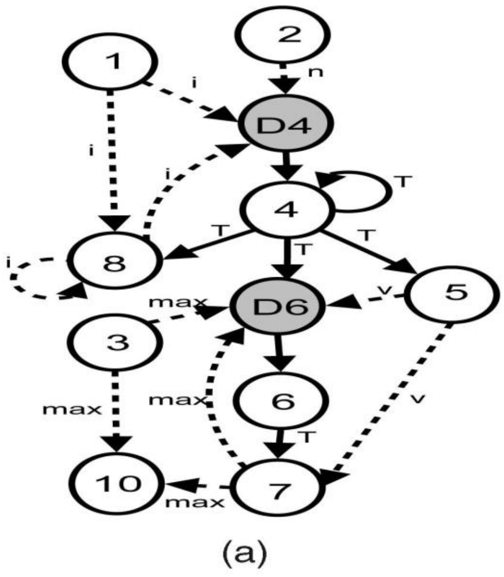


Fig 2: Structurally Transforming PDG

1.2 State specification

Our technique models states at each of the nodes after the PDG of the program has been structurally transformed.

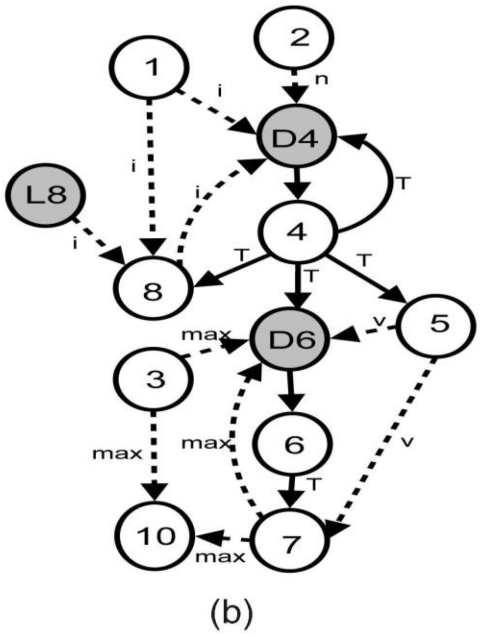


Fig 3. State Specification PDG

Shows the transformed PDG of find max. Predicate nodes. Our technique models the states at all predicate nodes in the transformed PDG using predicate outcomes. 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.,< ,> , _ , _ , ¼¼ , and 6¼). The technique places the simple predicates into two categories based on the state assignments.

1. For nodes whose predicates involve primitive variables(e.g., ðv1 relop v2Þ, where v1 and v2 are char, int, float, or double variables) and relop is a relational operator, the outcomes of the predicate computation are based on how v1 relates to v2. Represents a predicate "v > max" at node 6. If v ¼ 2and max ¼ 7 when node 6 is executed, the predicate outcome is <. In general, our technique assigns <, >,¼¼, and ? as the set of states to each predicate node whose operands are primitive variables.
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 ¼¼, 6¼ , and ? to the node.

Non predicate nodes.

Our characterization of the states of no predicate nodes that are dynamically data dependent on other nodes is based on a data-flow modelling technique proposed by Laski and Korel as a guide to program testing. Laski and Korel define the data environment of a statement s as the set of variable definitions that reaches s, along any paths, and is used at s. To more precisely model potential dynamic data flows, they introduced the concepts of elementary data context and data context for statements.

An elementary data context of a statement s is the set of definitions that reaches and is 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..

The set of states for a non predicate node that is dynamically data dependent on other nodes corresponds to the data context of that node, augmented with the? state. (Recall that means that the node was not executed in a given execution.)If a non predicate node is not dynamically data dependent on any node, then, by default, our technique assigns f>g as its data context. Hence, the states of the node are >and? The state > means that during a given execution, the node was executed. For example, nodes 1, 2, 3, 5, and L8 are not dynamically data dependent on any node.Hence, the nodes have the states. Shows the nodes in the transformed PDG with their corresponding states.

2. Learning

During this step, our technique estimates the parameters of the PPDG from the set of execution data (D ¼ fDkgnk¼0) generated by executing the instrumented program P0 with its test suite TP. Each Dk 2 D corresponds to a test case in TP. 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.

Technique uses node-state traces to estimate the parameters of the PPDG so that the PPDG will capture some of the temporal behaviours of the program. Each Dk 2 D is a node statetrace. A node can appear multiple times in the trace, and the states that the node assumes can be different. In this paper, we present a batch-learning algorithm. However, the algorithm can be modified easily to an online learning algorithm.

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 (CPTs), because the states of the nodes in the transformed PDG are discrete. Suppose X ¼ fX1; . . .; Xng denotes the set of nodes in the transformed PDG. We denote the it state associated with node Xj by xji the parents (immediate predecessors) of anode Xj by PaðXjÞ, and the it assignment of states to the parents of Xj by paji. For a node with no parents, our technique estimates the probabilities (pðXj ¼ xjiÞ) of the nodes aspðXj ¼ xjiÞ ¼. Where nðXj ¼ xjiÞ is the number of times node (Xj) is instates xji across all node-state traces and nðXjÞ is the number of times the node Xj occurs across all node-state traces.

For a node with parents, our technique estimates the probabilities (pðXj ¼ xjijPaðXjÞ ¼ pajiÞ) of the node as Where nðXj ¼ xji; PaðXjÞ ¼ pajiÞ is the number of times node Xj and its parents assume a specific state configuration across all node-state traces and nðPaðXjÞ ¼ pajiÞ is the number of times PaðXjÞ ¼ paji across all node-state traces.

A state configuration is a set of states assigned to a set of nodes in the PPDG. The CPTs of the nodes must satisfy (3), which means that the sum over the states of node Xj given that its parents are in a specific state configuration paji must equal 1.0:X

Learning algorithm (Learn Aram). Shows algorithm LearnParam that estimates the parameters of aPPDG. The algorithm takes as input a set of execution dataD, generated by executing an instrumented program P0with its test suite TP , and the program's transformed PDG. The algorithm outputs the PPDG of the program.

Learn-Param traverses each Dk 2 D from beginning to end, updating the parent states of nodes and the necessary counts depending on whether a node in a trace has parents. After Learn Aram processes D, it computes the conditional probabilities of each node in the transformed PDG. Finally, it returns the PPDG .the conditional probabilities distribution representations of each node in the transformed PDG of the example program (findmax). For example, the conditional probability distribution for node 6 in is denoted byPð6jD6Þ because node 6 is dependent on node D6. Thus, our technique estimates the probabilities of the states of node 6given the states of its parent node D6.

2.2 Learning Example

Suppose that the example program findmax receives the following inputs: ðn ¼ 1; v ¼ f1gÞ, ðn ¼ 2; v ¼ f1;_1gÞ,ðn ¼ 2; v ¼ f_1; 1gÞ, and ðn ¼ 1; v ¼ f0gÞ. These inputs cause findmax to execute correctly. Table 3 shows an example node-state trace. The first column shows the inputs tofindmax, where n is the number of inputs that findmaxreads at line 5 and v is the set of integers input into findmax.

Nodes in Transformed PDG with Corresponding Conditional Probability Distributions Nodes in Transformed PDG with Corresponding States estimate the probabilities in the conditional probability tables, LearnParam processes the traces from the beginning of the trace until the end, updating the states of nodes and their parent states. To illustrate, we show the estimation of the CPT for node 6. Note that node 6 is dependent on node D6 in the transformed PDG as shown in Fig. 4b. For the node-state trace , the first occurrence of node 6 has the state ">" and the state of node 6's parent at that occurrence is (d5ðvÞ, d3ðmaxÞ). Therefore, the algorithm increases nð6 ¼ ''>; '' D6 ¼ ðd5ðvÞ; d3ðmaxÞÞÞ by 1. Learn-Param continues processing the trace until it reaches the end. After all the traces have been processed, Learn Paramnormalizes the counts to produce the probabilities. Shows the conditional probability table for node 6. The first column shows the states of node D6 and the second column shows the states of node 6. The table shows thatPð6 ¼ ''>'' j D6 ¼ ðd5ðvÞ; d3ðmaxÞÞ ¼ 3=5, which means that the probability of node 6 assuming the state ">" given that node D6 has assumed the state ðd5ðvÞ; d3ðmaxÞ is 3/5. the sum of the probabilities for each row in the CPT for node 6 satisfies

III. APPLICATIONS OF THE PPDG

In this section, we apply the PPDG to two 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 we 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 suspicious.

FAULT LOCALIZATION

Debugging software is often a difficult and time-consuming task, which is mostly done manually. One of the most laborious aspects of debugging is fault localization—locating the faults in a program that caused one or more observed failures. To reduce the burden on the developer during fault localization, a number of fault localization techniques (e.g., [6], [14], [16], [18], [19], [26], [29], [31]) have been developed.

Existing fault localization techniques fall into two main categories: those that require knowledge of the incorrect values of program variables and those that do not require his knowledge. Techniques that require knowledge of incorrect variable values are mostly slicing techniques [29], [31]. The limitation of slicing techniques is that they do not provide an ordering or ranking4 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 difficulty of finding the faulty statement.

The techniques that do not require knowledge of incorrect variable values 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 [6], [26] 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 to be more effective in localizing faults than the second group. However, in CPT of Node 64. Informally, a ranking is an ordering among a set of elements according to a given criterion. practice, it is not always possible to have access to multiple executions that fail because of a given fault. Our fault localization algorithm (RankCP) is of the second type. Fault localization algorithm (RankCP). Fig. 6 shows algorithm RankCP, which analyzes a single failing execution at a time, and ranks nodes in the PPDG. RankCP ranks nodes based on the conditional probabilities of nodes given the states of their parent nodes (i.e., pðXj ¼ xjijPaðXjÞ ¼ pajiÞ), which reflect how the parents influence their children. Our hypothesis is that RankCP will often detect the first place in a failing execution, where a node (Xj) assumes an unusual state, given the states of its parents, thus indicating a possible cause of the failure. RankCP ranks a node Xj that has a state whose probability is low, given the states of Xj's parents, as highly suspicious. Our choice of this conditional probability as an inverse measure of suspiciousness is based on preliminary studies we conducted that showed that faults tend to be associated with low probability nodes. For a given program, RankCP inputs its PPDG and a node-state trace generated by a failing execution, and it returns a list of nodes ranked from most suspicious to least suspicious. Each node is also associated with a node-parent state configuration. RankCP processes a trace from beginning to end.

IV. SYSTEM DESIGN

Data Flow Diagram

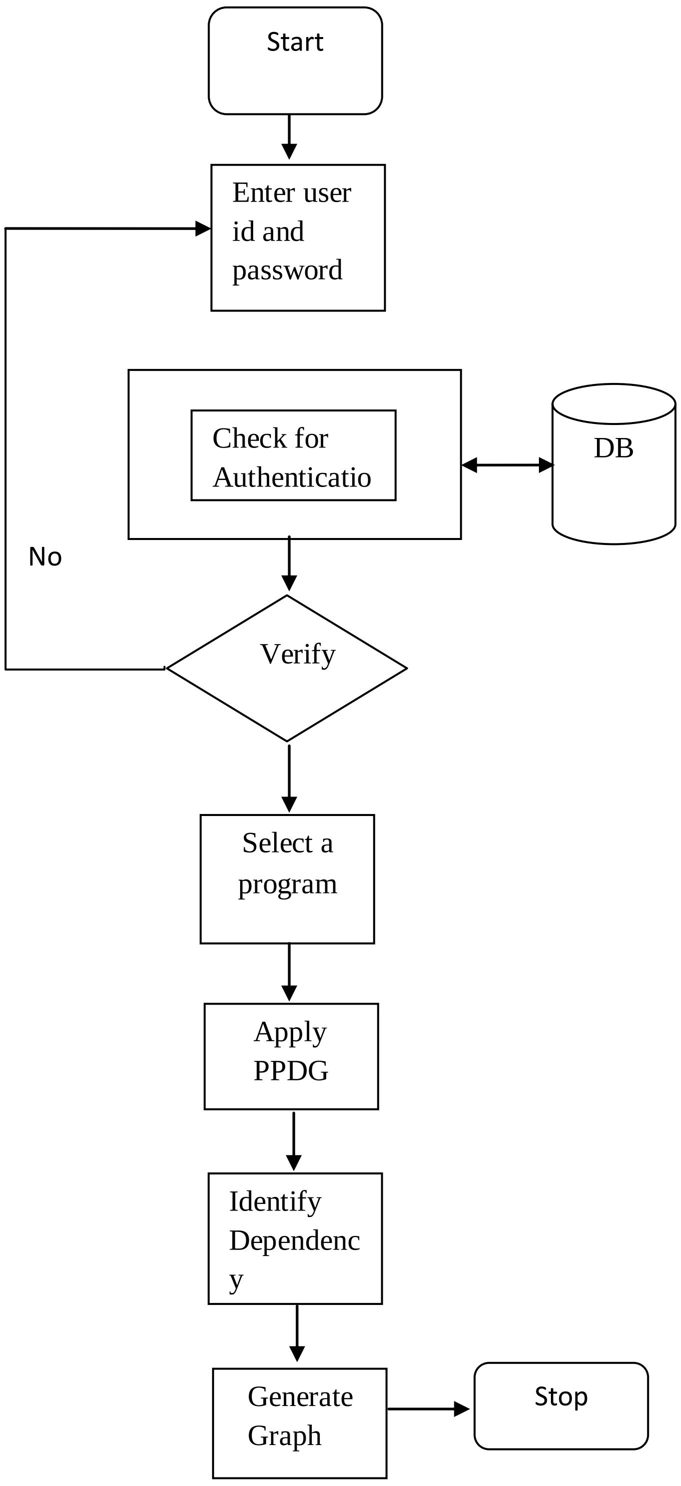


Fig 4. Data Flow Diagram

V. USE CASE DIAGRAMS

Use case diagrams model the functionality of system using actors and use cases.

User Login

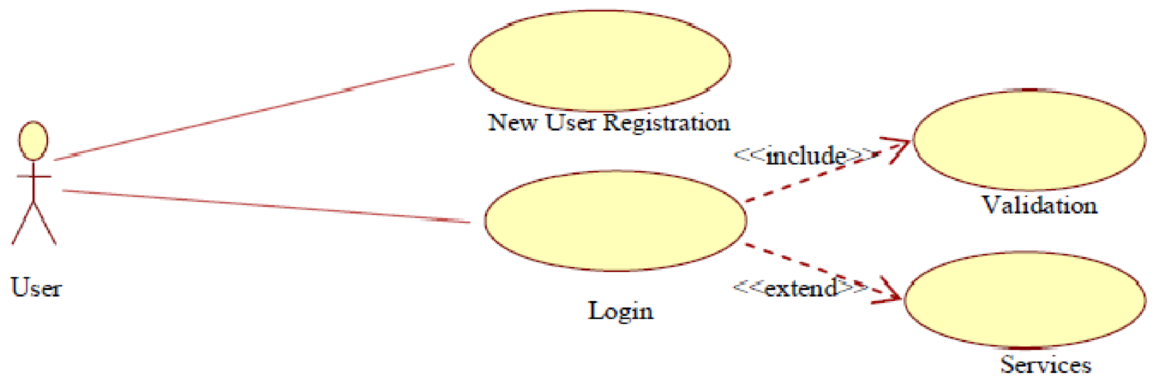


Fig 5.User Login

User Services

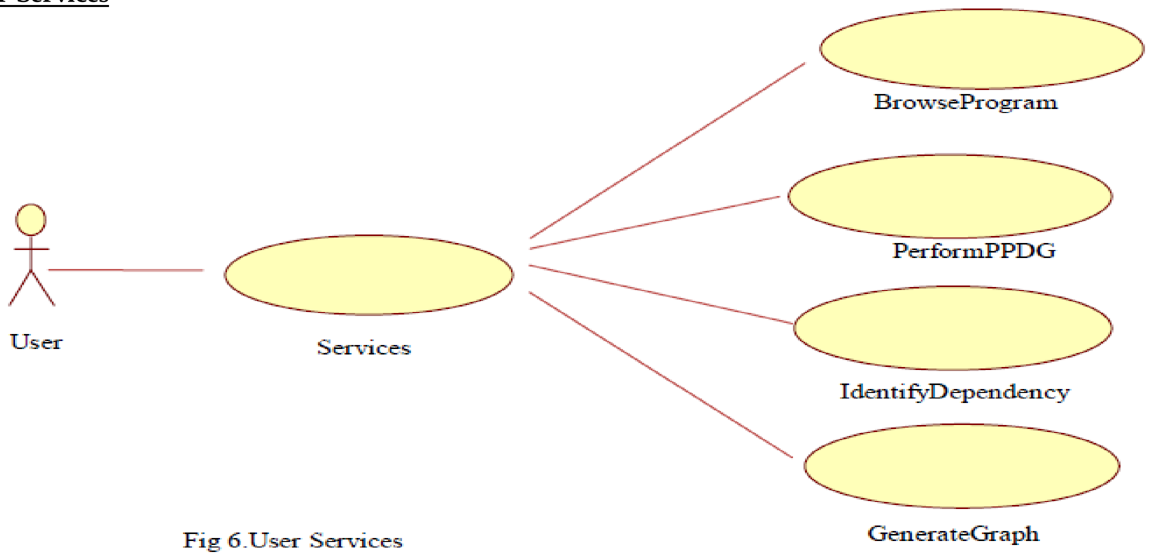
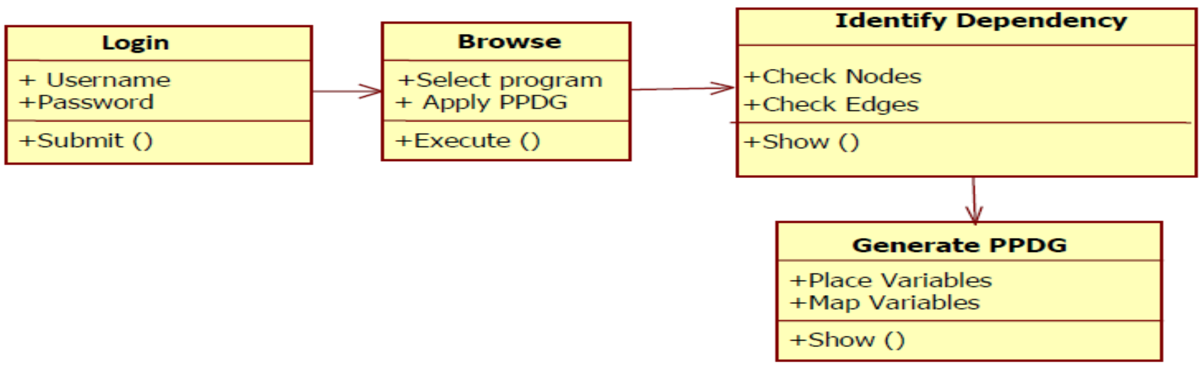


Fig 6.User Services

CLASS DIAGRAM



SEQUENCE DIAGRAM:

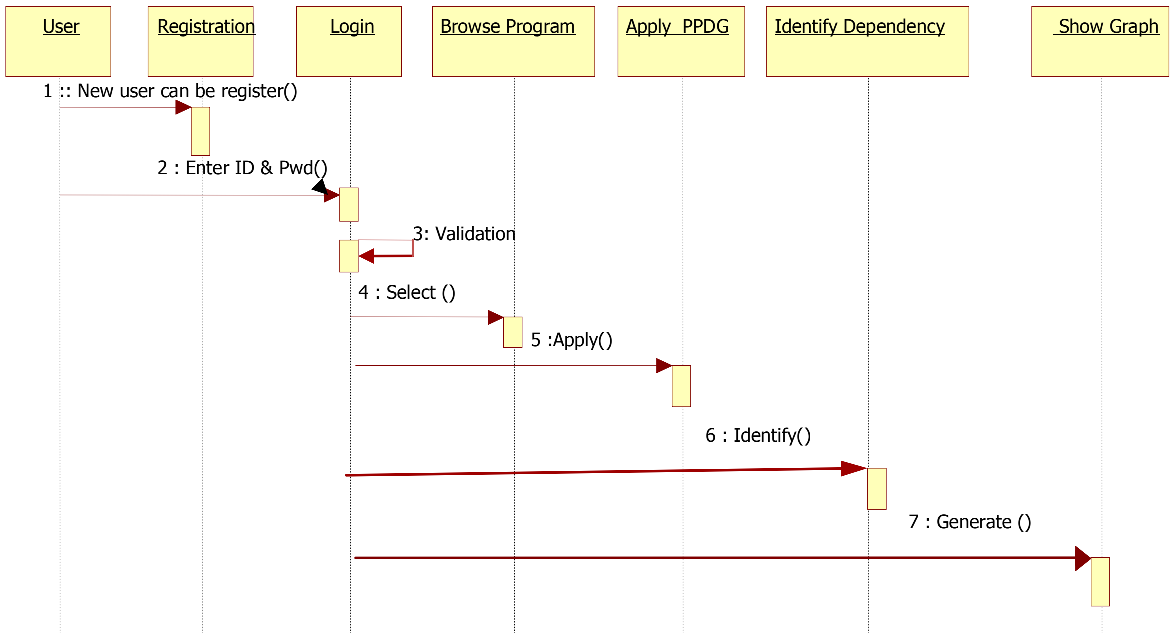


Fig 8.Sequence Diagram

VI. 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 behaviours. We also presented algorithms for two applications of the PPDG: which uses the PPDG to rank statements to assist in fault localization, and Fault-Comp, which uses the PPDG to generate explanations to aid in fault comprehension. The results also show that the PPDG can be an effective approximate model for representing behaviours of a program for fault diagnosis, eliminating the need to store large amounts of execution information during debugging.

The PPDG is based on the established framework of probabilistic graphical models, which are used widely in a variety of applications. This project presents algorithms for constructing PPDGs and applying them to fault diagnosis. The project also presents preliminary evidence indicating that a PPDG-based fault localization technique compares favourably with existing techniques. The project also presents evidence indicating that PPDGs can be useful for fault comprehension.

REFERENCES

[1]. R. Alur, P. _Cern_y, P. Madhusudan, and W. Nam, “Synthesis of Interface Specifications for Java Classes,” Proc. Symp. Principles of Programming Languages, pp. 98-109, Jan. 2005.
[2]. J.F. Bowring, J.M. Rehg, and M.J. Harrold, “Active Learning forAutomatic Classification of Software Behavior,” Proc. Int’l Symp.Software Testing and Analysis, pp. 195- 205, July 2004.
[3]. H. Cleve and A. Zeller, “Locating Causes of Program Failures,” Proc. 27th Int’l Conf. Software Eng., pp. 342-351, May 2005.
[4]. J. Ferrante, K.J. Ottenstein, and J.D. Warren, “The Program Dependence Graph and Its Use in Optimization,” ACM Trans. Programming Languages and Systems, vol. 9, no. 3, pp. 319-349, July 1987.
[5]. K.B. Gallagher and J.R. Lyle, “Using Program Slicing in Software Maintenance,” IEEE Trans. Software Eng., vol. 17, no. 8, pp. 751-761, Aug. 1991.