International Journal of Technical Research and Applications e-ISSN: 2320-8163,
www.ijtra.com Volume 3, Issue 2 (Mar-Apr 2015), PP. 260-268

DESIGN & DEVELOPMENT OF A PROBABILISTIC GRAPHICAL MODEL FRAMEWORK FOR PROGRAM BEHAVIOUR ANALYSIS

Prof. Dr. G. Manoj Someswar[1], Ms. Pushpanjali Patra[2]
[1]Professor & Dean (Research), [2]Associate Professor,
Department of CSE, Nawab Shah Alam Khan College of Engineering & Technology,
Affiliated to JNTUH, Malakpet, Hyderabad – 500024, Telangana, India.
manojgelli@gmail.com

ABSTRACT- This research paper presents an innovative model of a program's internal behaviour over a set of test inputs called the probabilistic program dependence graph (PPDG), which facilitates probabilistic analysis and reasoning about uncertain program behaviour 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 research paper presents algorithms for constructing PPDGs and applying them to fault diagnosis. The research paper also presents preliminary evidence indicating that a PPDG-based fault localization technique compares favourably with existing techniques. The research paper also presents evidence indicating that PPDGs can be useful for fault comprehension.

Keywords: Probabilistic Program Dependence Graph, Fault Localization, Dependency Network, regression testing, conditional statistical dependence.

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.[1] 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.[2] 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.[3] 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 this research 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.

II. THE ROLE OF PROBABILITY

It is important to distinguish two distinct ways in which probability enters into legal disputes. First, and increasingly, there are cases where the actual values of probabilities are or appear relevant to the issues, and estimates of them are given in testimony. These estimates will usually be based on statistical data, although they will inevitably also incorporate a range of assumptions, explicit or implicit, about the nature and relevance of such data.

For example, in a case where identification is based on a DNA match, a forensic scientist might testify that (on the basis of police DNA population samples and current genetic

understanding) the probability that a random individual might provide such a match is one in ten million; or, in the trial of a mother for murdering her babies, evidence might be offered, on the basis of an epidemiological survey,of the probability that they could have died from natural causes.[4] Such testimony is of course open to challenge on the grounds that the values given are wrong: they are based on wrong or irrelevant data, they have been calculated using inappropriate assumptions, their intrinsic uncertainty has been ignored, or purely speculative values have been treated as hard fact.This kind of attack is the bread and butter of the adversarial system,and as such can be readily understood,at least in broad outline, by the parties, the jury and the public.

III. OBJECTIVES

Exposes the natural framework about the program environment process. Identify the internal behaviour of the program. Identify the fault localization after deployment. Show the evidence for showing the dependence graph representation process. Identify statistical dependence identification process. Check the program behaviour and increased statically strength.

IV. CHALLENGES

Exposes the results in the form of graphical representation process. Graphical level information identifies the result of information in the form conditional dependence process.PPDG provides that information like abstract states representation process.[5] Information can be providing like valuable behaviour environment creation can be implemented inside the processing state.

V. 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.[6] 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.

VI. 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.

VII. DESIGN & DEVELOPMENT

A. Software Fault Localisation

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

Software fault localization is one of the most expensive activities in program debugging. It can be further divided into two major parts. The first part is to use a technique to identify suspicious code that may contain program bugs. The second part is for programmers to actually examine the identified code to decide whether it indeed contains bugs. All the fault localization techniques referenced in the following text focus on the first part such that suspicious code is prioritized based on its likelihood of containing bugs. Code with a higher priority should be examined before code with a lower priority, as the former is more suspicious than the latter, i.e., more likely to contain bugs. As for the second part, we assume perfect bug detection, i.e., programmers can always correctly classify faulty code as faulty, and non-faulty code as non-faulty. If such perfect bug detection does not hold, then the amount of code that needs to be examined may increase.

There is a high demand for automatic fault localization techniques that can guide programmers to the locations of faults with minimal human intervention. This demand has led to the proposal and development of various techniques over recent years. While these techniques share similar goals, they can be quite different from one another, and often stem from ideas which themselves originate from several different disciplines.

B. Bayesian Network

Bayesian networks are directed acyclic graphs whose nodes represent random variables in the Bayesian sense: they may be observable quantities, latent variables, unknown parameters or hypotheses.[7] Edges represent conditional dependencies; nodes which are not connected represent variables which are

conditionally independent of each other. Each node is associated with a probability function that takes as input a particular set of values for the node's parent variables and gives the probability of the variable represented by the node.

C. Hidden Markov Model

A hidden Markov model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (hidden) states. An HMM can be considered as the simplest dynamic Bayesian network. The mathematics behind the HMM was developed by L. E. Baum and coworkers.

In simpler Markov models (like a Markov chain), the state is directly visible to the observer, and therefore the state transition probabilities are the only parameters. In a hidden Markov model, the state is not directly visible, but output, dependent on the state, is visible. Each state has a probability distribution over the possible output tokens.[8] Therefore the sequence of tokens generated by an HMM gives some information about the sequence of states. Note that the adjective 'hidden' refers to the state sequence through which the model passes, not to the parameters of the model; even if the model parameters are known exactly, the model is still 'hidden'.

D. Dependency Network

The **dependency network** approach provides a new system level analysis of the activity and topology of directed networks. The approach extracts causal topological relations between the network's nodes (when the network structure is analyzed), and provides an important step towards inference of causal activity relations between the network nodes (when analyzing the network activity). This methodology has originally been introduced for the study of financial data, it has been extended and applied to other systems, such as the immune system, and semantic networks.

In the case of network activity, the analysis is based on partial correlations, which are becoming ever more widely used to investigate complex systems. In simple words, the partial (or residual) correlation is a measure of the effect (or contribution) of a given node, say *j*, on the correlations between another pair of nodes, say *i* and *k*. Using this concept, the dependency of one node on another node, is calculated for the entire network. This results in a directed weighted adjacency matrix, of a fully connected network. Once the adjacency matrix has been constructed, different algorithms can be used to construct the network, such as a threshold network, Minimal Spanning Tree (MST), Planar Maximally Filtered Graph (PMFG), and others.

E. Software Design

The objects discovered during analysis can serve as the design or framework. A DFD is a graphical tool used to describe and analyze the movement of the data through a system including the process, stores of data, and flows in the system. Next,we focus on the Uml diagrams.

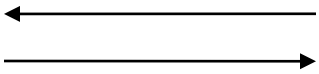
VIII. DATA FLOW DIAGRAM

A DFD is a graphical tool used to describe and analyze the movement of the data through a system including the process, stores of data, and flows in the system. A DFD is also known as "Bubble Chart" has the purpose of clarifying system requirements and identifying major transformations that will be used in system design.

DFD Symbols: In the DFD there are four symbols.

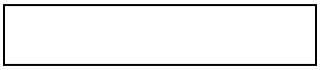
1. A square defines a source or destination of the system data.

2. An arrow identifies data flow. It is the pipeline through which the information flows.



3. A circle or bubble represents a process that transforms incoming data flow into outgoing data flows.

4. An open rectangle is a data store, data at rest or a temporary repository of data.



Data Flow Diagram

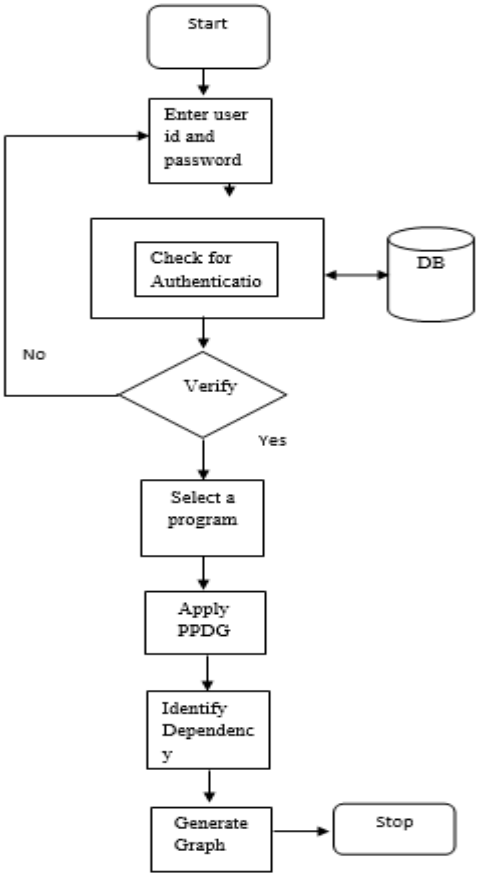


Figure 1: Data Flow Diagram

A. Uml Diagrams

Unified Modelling Language (UML) is a Standardized notation for object- oriented analysis and design UML is a general-purpose modelling language that includes a graphical notation used to create an abstract model of a system, referred

to as a UML model. Structure Diagram emphasize what things must be in the system being modelled: such as Class diagram.

Behaviour Diagram emphasizes what must happen in the system being modelled:

- Use case diagram
- Class diagram
- Activity diagram
- Sequence diagram

B. Use Case Diagram

A use case is a set of scenarios that describing an interaction between a user and a system. A use case diagram displays the relationship among actors and use cases. The two main components of a use case diagram are use cases and actors. An actor is represents a user or another system that will interact with the system you are modelling. A use case is an external view of the system that represents some action the user might perform in order to complete a task.

C. Class Diagram

In the Unified Modeling Language (UML), a class diagram is a type of static structure diagram that describes the structure of a system by showing the system's classes, their attributes, and the relationships between the classes.

D. Activity Diagram

It describes the workflow behaviour of a system. Activity diagrams are similar to state diagrams because activities are the state of doing something. The diagrams describe the state of activities by showing the sequence of activities performed.[9] Activity diagrams can show activities that are conditional or parallel.

E. Sequence Diagram

A sequence Diagram is an interaction diagram that emphasizes the time ordering of message. It shows a set of objects and the messages that sent and received by those objects.

An object in a sequence diagram is rendered as a box with dashed line descending from it. The line is called the object lifeline, and its represents the existence of an object over a period of time.

The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action.

F. Collaboration Diagram

Collaboration diagrams represent interaction between objects as a series of sequenced messages. Unlike a sequence diagram, we don't have to show the lifeline of an object in a collaboration diagram. The sequences of objects are indicated by sequence numbers preceding messages.

G. Use Case Diagrams

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

User Login

.

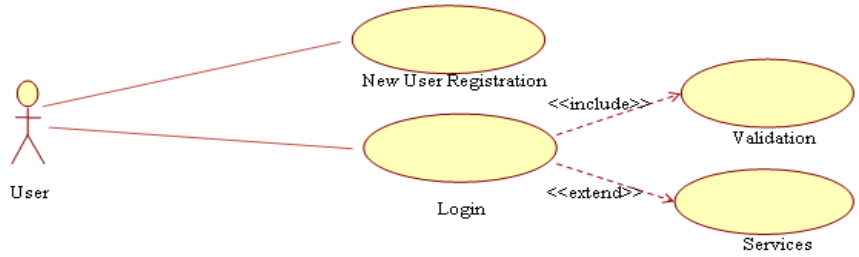


Figure 2: User Login

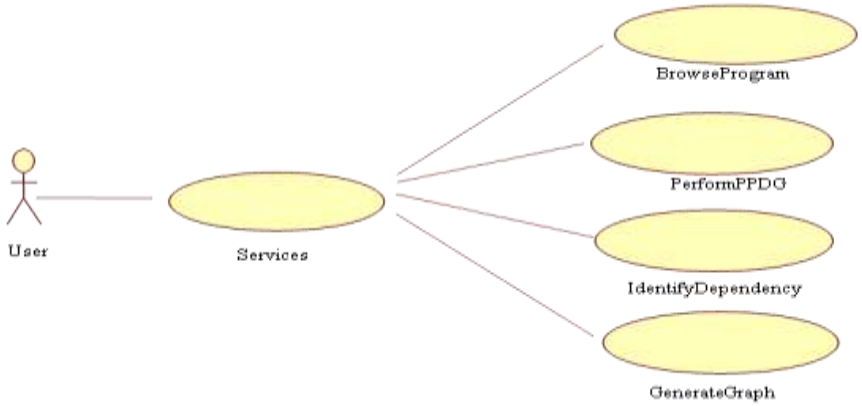


Figure 3: User Services

A. Class Diagram

Class diagrams that shows a collection of static model elements such as classes, types, and their contents and their

relationships. Class is a set of objects that share same attributes, operations, and relationships A class is represented as a rectangle. Classes are arranged in hierarchies by sharing common structure and behaviour,that are associated with other classes.

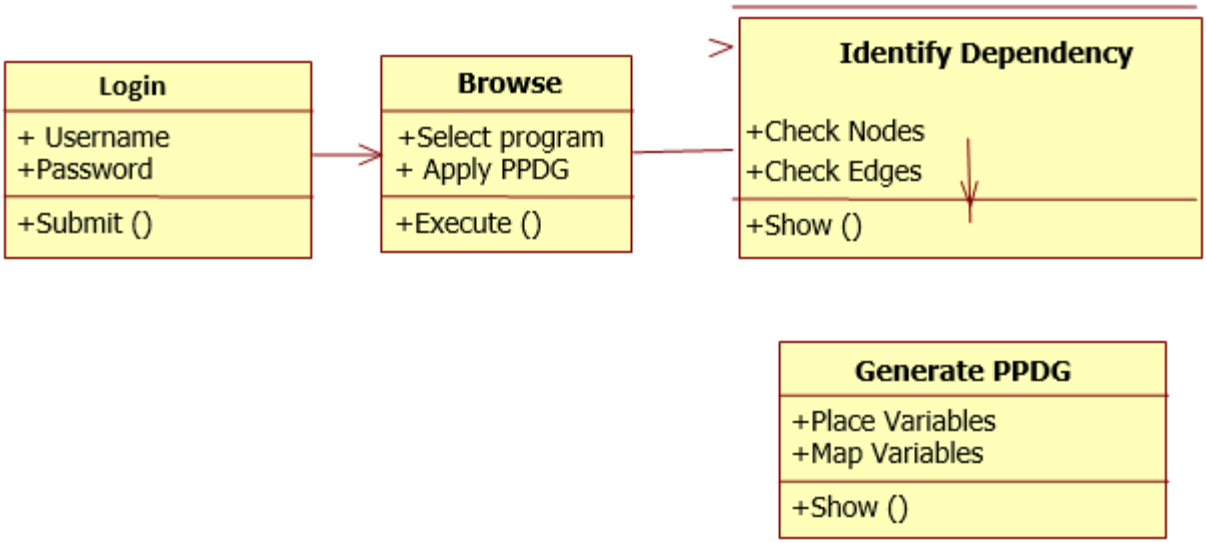


Figure 4: Class Diagram

B. Sequence Diagram:

A sequence diagram is an interaction diagram that emphasizes the time ordering of messages. It shows a set of objects and the messages that sent and received by those objects. An object in a sequence diagram is rendered as a box with a dashed line descending from it. The line is called the object lifeline, and it represents the existence of an object over a period of time. Messages are rendered as horizontal arrows being passed from object to object as time advances down the object lifelines indicates that message gets passed. The focus of control is a tall, thin rectangle that shows the period of time during which an object is performing an action.

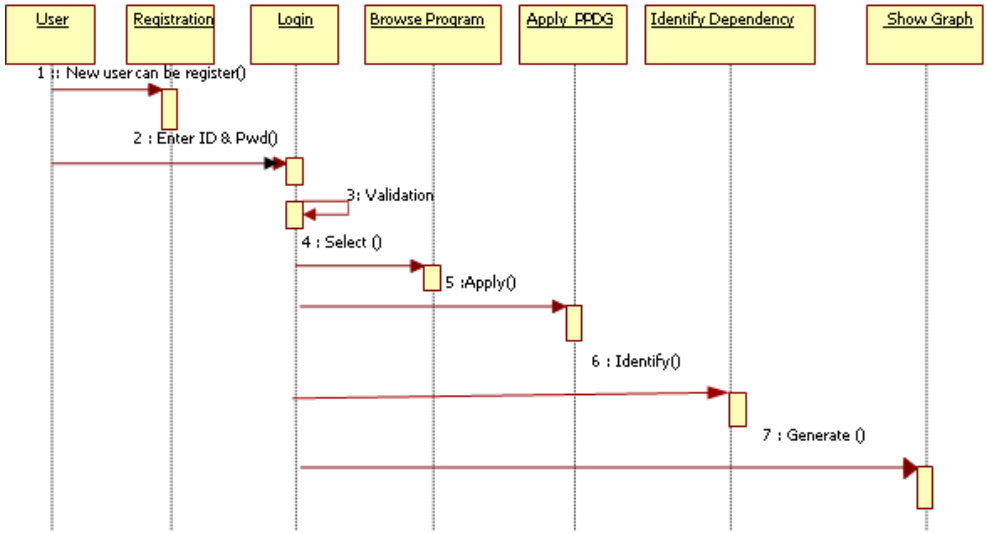


Figure 5: Sequence Diagram

C. Collaboration Diagram

Collaboration diagrams represent interactions between objects as a series of sequenced messages. Unlike a sequence diagram, we don't have to show the lifeline of an object in a collaboration diagram. The sequences of objects are indicated by sequence numbers preceding messages.

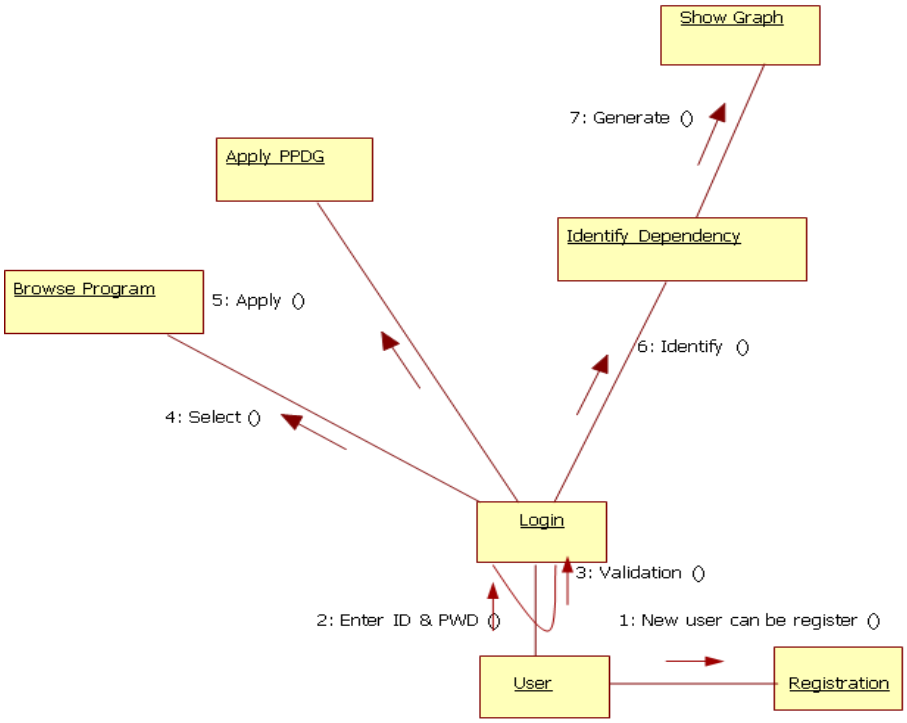


Fig: 4.6 collaboration diagram

D. Activity Diagram:

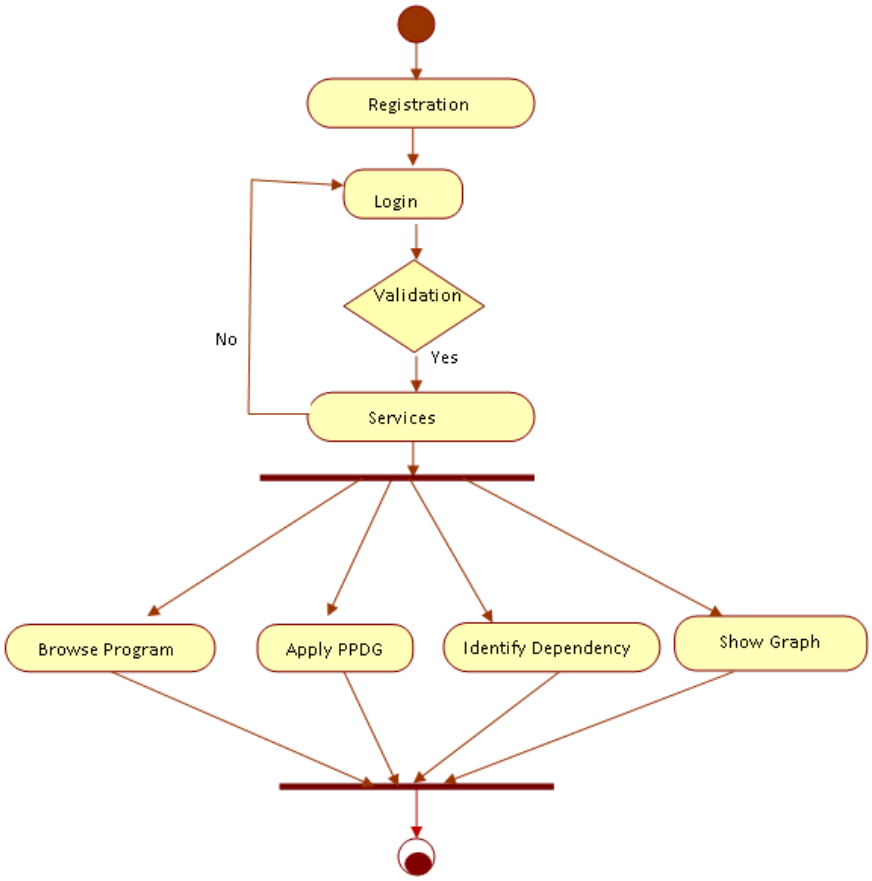


Figure 6: Activity Diagram

Activity diagram is another important diagram in UML to describe dynamic aspects of the system. Activity diagram is basically a flow chart to represent the flow from one activity to another activity. This flow can be sequential, branched or concurrent. Activity diagrams deals with all type of flow control by using different elements like fork, join etc. There can be only one start state in a activity diagram, but there may be many final states.

IX. SOFTWARE TESTING

Software Testing is the process used to help identify the correctness, completeness, security and quality of developed computer software. Testing is a process of technical investigation, performed on behalf of stakeholders, that is intended to reveal quality-related information about the product with respect to the context in which it is intended to operate.

The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It provides a way to check the functionality of components, sub assemblies, assemblies and/or a finished product It is the process of exercising software with the intent of ensuring that the Software system meets its requirements and user expectations and does not fail in an unacceptable manner. There are various types of test. Each test type addresses a specific testing requirement.

TESTING METHODOLOGIES

- **Black box Testing**: is the testing process in which tester can perform testing on an application without having any internal structural knowledge of application. Usually Test Engineers are involved in the black box testing.
- **White box Testing**: is the testing process in which tester can perform testing on an application with having internal structural knowledge. Usually The Developers are involved in white box testing.
- **Gray Box Testing**: is the process in which the combination of black box and white box tonics' are used.[10]

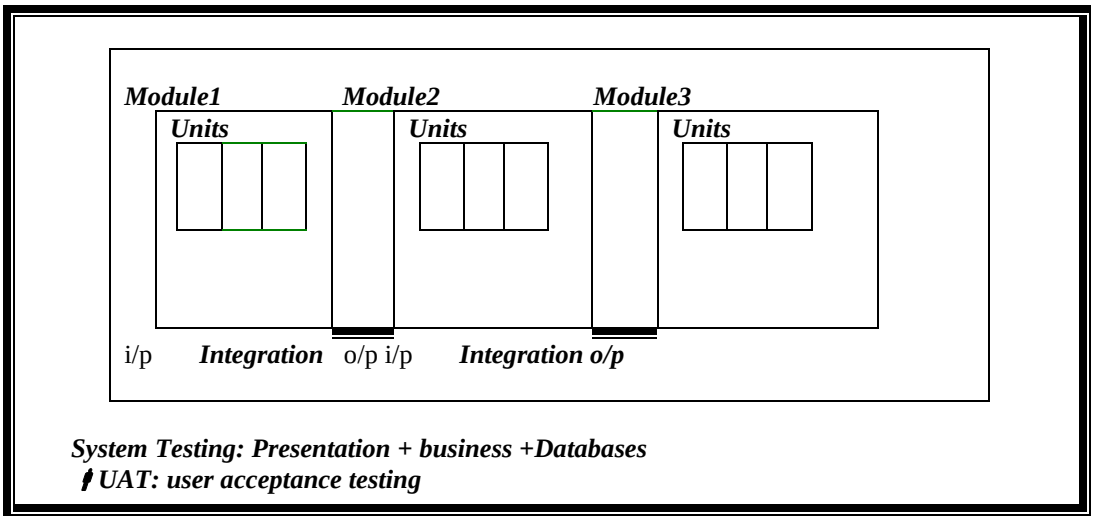


Figure 7: Levels of Testing

TESTING STAGES

Unit testing: Unit testing involves the design of test cases that validate that the internal program logic is functioning properly, and that program input produces valid outputs. all decision branches and internal code flow should be validated .it is the testing of individual units before integration .It is done after the completion of an individual unit before integration. This is a structural testing,that relies on knowledge of its construction and is invasive. Unit tests perform basic tests at component level and test a specific business process, application and/or system configuration. Unit tests ensure that each unique path of a business process performs accurately to the documented specification and contains clearly define inputs and expected results.

Integration testing: Integration tests are designed to test integrated software components to determine if they actually run as one program. Testing is event driven and is more concerned with the basic outcome of screens or fields. Integration tests demonstrate that although the components were individually satisfaction, as shown by successfully unit testing, the combination of components is correct and consistent. Integration testing is specifically aimed at exposing the problems that arise from the combination of components

Integration testing is of three types.

- Bottom up integration
- Top down Integration
- Sandwich Integration

Bottom up integration testing consists of unit testing followed by system testing. Unit testing has the goal of testing individual modules in the system. subsystem testing is concerned with verifying the operation of the interfaces between modules in the sub systems. Top down integration testing starts with the main routine and one or two immediately subordinate routine in the system structure. top down integration requires the use of program stubs to simulate the effects of lower levels runtimes that are called by those being tested

Functional testing: Functional tests provide a systematic demonstrations that functions tested are available as specified by the business and technical requirements, system documentation, and user manuals. Functional testing is cantered on the following items:

Valid Input: identified classes of valid input must be accepted.

Invalid Input: identified classes of invalid input must be rejected.

Functions: identified functions must be exercised.

Output: identified classes of application outputs must be exercised.

Systems/Procedures: interfacing systems or procedures must be invoked.

Organization and preparation of functional tests is focused on requirements, key functions, or special test cases. In addition, systematic coverage pertaining to identify Business process flows; data fields, predefined processes, and successive processes must be considered for testing. Before functional testing is complete, additional tests are identified and the effective value of current tests is determined.

System Test: System testing ensures that the entire integrated software system meets requirements. It tests a configuration to ensure known and predictable results. An example of system testing is the configuration oriented system integration test. System testing is based on process descriptions and flows, emphasizing pre-driven process links and integration points.

X. TYPES OF TESTING EMPLOYED

A. Smoke Testing

Is the process of initial testing in which tester looks for the availability of all the functionality of the application in order to perform detailed testing on them.

B. Regression Testing

Is one of the best and important testing. Regression testing is the process in which the functionality, which is already tested before, is once again tested whenever some new change is added in order to check whether the existing functionality remains same.

C. Static Testing

Is the testing, which is performed on an application when it is not been executed. Ex: GUI, Document Testing.

D. Dynamic Testing

Is the testing which is performed on an application when it is being executed. Ex: Functional testing.

E. Alpha Testing

It is a type of user acceptance testing, which is conducted on an application when it is just before released to the customer.

F. Beta-Testing

It is a type of UAT that is conducted on an application when it is released to the customer, when deployed in to the real time environment and being accessed by the real time users.

G. Compatibility Testing

It is the testing process in which usually the products are tested on the environments with different combinations of databases (application servers, browsers…etc) In order to check how far the product is compatible with all these environments platform combination.

H. Installation Testing

It is the process of testing in which the tester try to install or try to deploy the module into the corresponding environment by following the guidelines produced in the deployment document and check whether the installation is successful or not.

TEST CASES

POSITIVE TEST CASE

S .No	Test case Description	Actual value	Expected value	Result
1	Create the new user registration process	New user created successfully	To update the database in MSACESS	True
2	Browse/Select one program	Selected a program from the folder/file	Execute the program	True
3	Identify the Dependency variables	Identified the dependency information from the program	Shows the dependency variables	True
4	Generate Graph.	Depending on the dependency show the graph	Show the graph	True

Table 1: Positive Test Case

NEGATIVE TEST CASE

S .No	Test case Description	Actual value	Expected value	Result
1	Create the new user registration process	Invalid User	Cannot update the data in the database MSACESS	False
2	Browse/Select one program	Cannot select a program from the folder/file	Execution Fails	False
3	Identify the Dependency variables	Does not exist any dependency information	Shows no dependency variables in the program	False
4	Generate Graph.	No dependency no graph	Do not generate the graph	False

Table 2: Negative Test Case

XI. RESULTS & CONCLUSION

In this research 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.

XII. FUTURE ENHANCEMENTS

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.

[6]. D. Heckerman, D.M. Chickering, C. Meek, R. Rounthwaite, and C.M. Kadie, "Dependency Networks for Inference, Collaborative Filtering, and Data Visualization," J. Machine Learning Research,vol. 1, pp. 49-75, 2000.

[7]. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow and Controlflow-Based Test Adequacy Criteria," Proc. Int'l Conf. Software Eng., pp. 191-200, May 1994.

[8]. J.W. Laski and B. Korel, "A Data Flow Oriented Program Testing Strategy," IEEE Trans. Software Eng., vol. 9, no. 3, pp. 347-354, May 1983.

[9]. W. Masri and A. Podgurski, "An Empirical Study of the Strength of Information Flows in Programs," Proc. 2006 Int'l Workshop Dynamic Systems Analysis, pp. 73-80, 2006.

[10]. K. Murphy, "Dynamic Bayesian Networks: Representation, Inference and Learning," PhD thesis, Computer Science Division, Univ. of California, Berkeley, 2002.