An Algorithmic Approach to Predict Fault Propagation and Defects in Dependent Modules based on Coupling

Kireet Joshi
M.Tech Scholar, Computer Science & Engineering
B.T.K.I.T, Dwarahat

Ramesh Chandra Belwal
Asst. Professor, Deptt. Of Computer Science & Engineering
B.T.K.I.T, Dwarahat

Shailendra Mishra, PhD.
Prof. & H.O.D, Computer Science & Engineering
B.T.K.I.T, Dwarahat

ABSTRACT
There is an enormous amount of research going on to minimize the effect of coupling between the software modules and to reduce the defects present in them. In this paper, an algorithmic approach is proposed that gives a probability, such that the highly dependent modules in system must be analyzed by the development team for fault proneness and defects. The higher the coupling, interdependency between the modules is increased and it is alarming issue in software engineering tasks. There is an enormous amount of research done on direct and indirect coupling, but this paper approaches on the effect of coupling to predict defects and how they are propagating between the modules. Every software product is tested for defects and bugs before it is given to acceptance testing to users. The paper focuses on testing the defect propagation percentage of every module in a dependent system (dependent modules).The greater the percentage of defect propagation factor between two dependent module, implies that the coupling between them is higher and the probability of the module to be fault prone increases. Taking this into consideration, the testing team saves the time by considering more on the modules for which the percentage defect propagation factor is higher. It ensures time, cost and efficiency which are the main factors of a software industry.

Keywords
Coupling, Fault detection, Fault Prediction using Coupling, Module Dependency, Testing Strategies, Fault Localization, Defects, Debugging

1. INTRODUCTION
Dependency between the software modules is an issue in software engineering tasks, but if the dependency is higher than at the time of testing the modules the tasks become cumbersome. In software engineering, coupling or dependency means each program module relies on each one of the other modules. Coupling is usually contrasted with cohesion. Low coupling often correlates with high cohesion, and vice versa. Low coupling is often a sign of a well-structured computer system and a good design, and when combined with high cohesion, supports the general goals of high readability and maintainability. Coupling was introduced in software engineering tasks for designing the modules. It was observed by the developers that there are some programs that were easier to implement in software processes , and coupling in software engineering is a method of finding how independent a proposed module or node is there from others in the software system[17][18].The main idea of incorporating coupling in program modules is to minimize the cost and time of "debugging". Coupling gives an idea and concept of the strength of interconnections between program units. Highly coupled system contains program units dependent on each other. Loosely coupled modules constitute some program units that are independent or almost independent. There are some program or software modules that are not correlated to each other if they can function completely without the interference of the other. Obviously, there can't be any modules that completely independent of each other. They must interact so that desired outputs can be produced. If connections between modules increase abruptly, then the chances of dependency in the modules increase in the sense that, more information of one module is required to understand the characteristics of other module. There are three factors that the programmer should be known of like, number of interfaces, complexity of interfaces and type of information flow along interfaces. If the programmer wants to minimize number of interfaces between modules, he should make an attempt to minimize the complexity of each interface, and control the type of information flow. An interface of a module is used to pass information to and from other modules. In general, modules are tightly coupled if they use shared variables or if they exchange control information. Loose coupling in module means information held within a unit and interface with other units via parameter lists. Tight coupling within module suggest shared global data. There are two types of info flow in modules, data or control. Passing or receiving back control info means that the action of the module will depend on this control info, which makes it difficult to understand the module. Interfaces with only data communication result in lowest degree of coupling, followed by interfaces that only transfer control data.

2. MODULE DEPENDENCY
It means when some of the programs modules are directly or indirectly related or dependent on other modules. While doing programming tasks the developers uses some slices which are used to investigate the features of the dependencies in a program. Dependencies within a program module should be checked, and more focus should be given so that in near future it can be used in many software engineering tasks like debugging and fault localization or defect prediction within modules of a program and it gives the tester an ease to easily predict and analyze the results based on the focused modules. A number of Static program-dependency models existing in today's technological computing era , like the program-dependence graph [16] and the system-dependence graph [15], that are being used in various software engineering methodologies and also been used by many researchers and programmers for incorporating the techniques in supporting software-engineering tasks such as debugging, testing and maintenance(e.g., [13, 14]).

2.1 STATIC AND DYNAMIC MODELS
The program-dependence graph (PDG) and system-dependence graph (SDG) are static program models. They are used to determine which instructions in a program are related to the other instructions. Dynamic slices only represent the

events that actually occurred during a specific execution, thus removing the instructions that were not relevant for that execution. In this case the programmer is more concerned with only on the specific instruction that are taking part in the execution process[1].The PDG is an intra procedural model of a procedure that captures both control and data dependencies among instructions within the procedure.

2.2 Control and Data Dependency

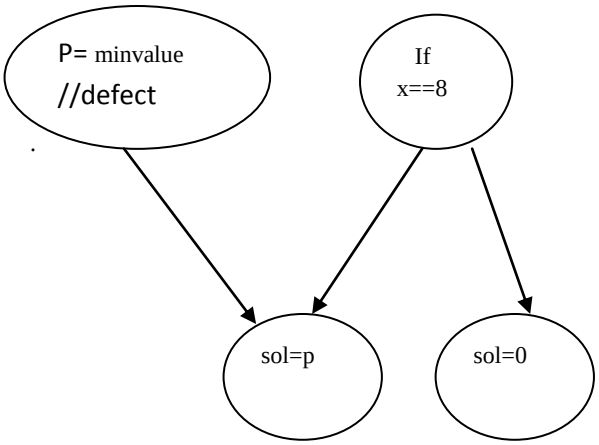

Fig 1: Diagram Depicting Data and Control Dependencies between Different Modules

Data coupling is the coupling in which one software module transfers the information to another module for its future tasks. Control coupling is the coupling in which output of one module depends on the successful execution of the module on which it is dependent. In the above figure, the value of P is used by sol shows data dependency or coupling, where the output of one component or module can be used as an input to another.Similarly,successful execution of the module(x==8)will ensure whether the two modules will be execute or not, this accounts for control coupling.

3. FAULT LOCALIZATION

Detection of software fault by runtime monitor or by any testing tool, the fault localization process is very difficult and costly and requires enormous efforts of the developer and the analyst [9].Software fault localization is very costly and time consuming issue in module or component debugging. So, research is going for some fault localization techniques that can guide software developers to localize the faults in a module with minimum time and effort. This research has focused to the proposal and development of various methods, each of which seeks to make the fault localization process more effective in its own unique and creative way. To improve the quality of a module or a component, one have to remove as many defects in the program as possible without introducing new bugs at the same time. During module or component debugging, fault localization is the phenomenon of identifying the exact locations module for faults. It is a very expensive and time consuming process. Its effectiveness depends on software developers understanding of the module being debugged, and their ability of logical judgment, past experience in program or component debugging, and how suspicious code, in terms of its likelihood containing faults, is identified and prioritized for an examination of possible fault locations. When the software faults are detected in some modules or programs then there should be some fault localization algorithms that can be incorporated to minimize the manual inspection and operational cost [4].Fault localization can be categorized into two parts. The first part is to use a method to identify suspicious code in a program module that may contain program bugs. The second part is for software developers to actually examine and analyze the identified code to decide whether it indeed contains bugs. All the fault localization methods referenced in the following focus on the first part such that suspicious code is prioritized based on its likelihood of containing bugs. Software module or a program code containing higher priority should be analyzed before the software module or program code with a lower priority, as the former is more suspicious than the latter, i.e., more likely to contain bugs. If the testing team members are able to analyze the source module from where the faults are propagating to the dependent modules then a lot of time and effort can be saved and detection of faults or defects in early stages helps the testing team members to analyzed the affected module only from where the defect is being propagated irrespective of analyzing or testing the overall system. As for the second part, assume perfect bug detection. A bug in a piece of software module or a program code will be analyzed by a developer if the module (namely, the statement) is analyzed and examined thoroughly. If the developer is unable to analyze perfect bug detection, then the software module or the program code (the number of statements in this case) needs to be examined in order to find the bug. Without any loss of information, the program module may be referred to as statements having the understanding that, fault localization methods can also be applied in order to identify the suspicious modules, decisions, definitions etc.Program slicing is used for debugging the software modules which comprises of the overall computation of program statements [11,12].

4. DEFECTS

While testing when a tester executes the test cases, he might observe that the actual test results do not match from the expected results. The variation in the expected and actual results is known as defects. Different organizations have different names to describe this variation, commonly defects are also known as bug, problem, incidents or issues. The cost of searching the defects is a cumbersome task and is included as one of the most expensive software development tasks. But the effect of defects can be minimized by incorporating some defect management process that focuses on reducing the impact of defects in software engineering tasks. There have been various research in past that showed that defect prediction models that are built on some kind of product metrics, and can be used to improve the quality of software packages or modules [2].There may be a chance that there are some incidents that occurs during testing may not be a defect or bug.TDD (TEST DRIVEN DEVELOPMENT) is a relatively new software development practice that have been developed in such a way by the developers, in which they write the unit tests before coding of program starts. Analysing and detecting the software bugs before implementation of the actual code is assumed to be much cheaper and time saving, than after implementation of the same [6].

Test-Driven Development (TDD) is an approach in software engineering that consists of preparation of very short iterations where the test case(s) covering a new feature or

functionality are written first. Defects are fixed and all the suitable components are restructured to finalize the changes.

5. TESTING

Software Testing is the process of executing modules or system with the intent of finding errors. Testing different modules or a component is one of the most common methods for assuring quality of complex computer software systems. To be confident of the results of testing, testers need for-manually defined procedures that provide mechanisms for creating test data and for deciding when testing can stop. Test requirements are specific things that must be satisfied or covered during testing. Many Researchers have opted some important issues regarding the test cases. How can each additional failed test cases can help in locating the modules for bugs and defects and simultaneously how an additional successful test case helps the programmers in locating bugs in software modules[7]. A testing criterion is a rule or collection of rules that impose requirements on a set of test. The paper focuses on finding the probability of a defect prone module; so that it will save the time and cost factors that are important issues in terms of software industry. The first phase of testing is done by the designers and engineers who created the system, usually before the system is delivered to the customer. The test data that is used in this first phase is similar to data that would be used by the actual customer. The second phase of testing is done after the system has been delivered and installed with the customer. The data used in the second phase is usually 'live' data - data that is actually part of the customer's organization. Because of programmers competence limit, there exist enormous amount of defects that are generated during the software development life cycle in every software engineering tasks [8].

6. RELATED WORK

Research has been done in past years to understand the dependencies among the program elements, among multiple modules. The concept of weighted system dependence graph has been proposed to account for the propagating faults among the dependent modules. The weight for a particular dependency edge, based on co-execution, is defined by the degree to which the same executions executed each of its incident nodes. Past research has been done to calculate the degree to which the same amount executions were performed in each of the dependent program modules by using some mathematical function. Calculating the degree of executions by incorporating the Jaccard similarity coefficient can be used in this perspective [10].The co-execution edge weight is calculated using the following formula

$$Jaccard(Ei_1, Ei_2) = \frac{|Ei_1 \cap Ei_2|}{|Ei_1 \cup Ei_2|} \quad (1)$$

Where i1 and i2 are two instructions with a dependency between them, and Ei1 and Ei2 are the sets of executions that executed i1 and i2, respectively. Past work has to done to calculate the direct and indirect coupling within the modules that are dependent to each other, but there still there may be hidden dependencies. The longer the two modules are connected to each other the more hidden dependence. Indirect coupling can also be analyzed and detected by the transitive closure of the modules, but there may be some circumstances that instead of transitive closure the indirect coupling may still exist and leads hidden modules undetected in software engineering process[3].

The Coupling Metrics can be measured by the equations:

$$ICM = \frac{\sum ICi}{\sum C_i} \quad (2)$$

Where ICi is the number of classes to which a class Ci is indirectly coupled, and the summation is over all the classes.

$$DCM = \frac{\sum DCi}{\sum C_i} \quad (3)$$

Where DCi is the number of classes to which a class Ci is directly coupled, and the summation is over all the classes.

$$failurecorrelation = \frac{\boldsymbol{failed\ (k)}}{\sqrt{\boldsymbol{totalfailed} * (\boldsymbol{failed(k)} + \boldsymbol{passed(k)}}} \quad (4)$$

Where, failed (k) is the number of failed test cases or defect producing variables between dependent modules.
Passed (k) is the number of passed test cases or the variables which are not producing defects between dependent modules
Totalfailed are the total number of defect producing variables between dependent modules [19].
Existing Algorithm calculates the failure- correlation as a measure of detecting the fault prone modules. Higher the failure-correlation percentage means that the dependent modules are prone to fault and testing team have to focus more on those modules. The proposed algorithm calculates the Defect Propagation factor to find the probability of the dependent modules to be fault prone.

7. Proposed Approach

Some abbreviations that are used throughout in the paper for calculations:

Calculated Factor CFij is given by

$$\frac{\boldsymbol{common\ variables\ between\ dependent\ modules}}{\boldsymbol{total\ variables\ in\ dependent\ modules}}$$

And Percentage Defect propagation factor DFij is given by

$$\frac{\boldsymbol{defect\ producing\ variables\ participating\ between\ dependent\ modules}}{\boldsymbol{common\ variables\ between\ dependent\ modules}}$$

If the defect for which the programmer is applying the effort is a real defect or bug, then the programmer should focus only on the specific executions instead of detecting all possible executions in a module. Hence there is an increased probability of the bug being detected in the program [5].In the proposed approach there is intent of finding the probability of a module to be more fault prone.

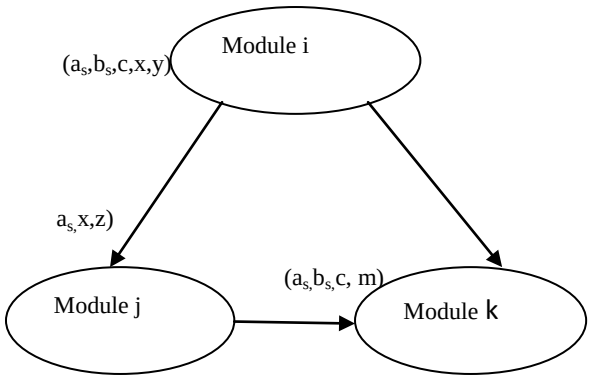

Fig 2: Dependency between modules

Suppose from the above figure, there is a set of interlinked modules such that one of the modules is interdependent on another. Let CDij consists of common variables between modules i and j.CDik consist of common variables between modules i and k.CDjk consist of common variables between modules j and k.Let DVij be the set of defect producing variables participating between dependent modules(i and j).DVjk be the set of defect producing variables participating between dependent modules (j and k). The intent is to find the probability of the module to be fault prone.

If the Calculated Factor CFij, exceeds the percentage defect propagation factor DFij, then it can be said that the module having high percentage defect propagation factor with respect to Calculated Factor CFij is statistically more fault prone, and the dependency (interdependency) is higher. This accounts for high coupling. So it will help the testing team to focus more on that defected module for further debugging and their efficiency increases.

In the above figure, module i is having variables set (a_s,b_s,c,x,y) where a_s,b_s are the variables producing defects**.** As it is clear from the figure that module k contains both of the defect producing variables a_s,b_s as compared to module j which consists of only one defect producing variable a_s.So,there is a probability that module k which is dependent on module i will be more fault prone then module j which is also dependent on module i.

7.1. Proposed Algorithm

Input –

All variables in each module.

All defect producing variables in each module.

Output: Set of the dependent defect prone modules

Method:

For each module Mi, get the set of dependent modules // where i=1 to n

For each module Mi, Where i=1 to n

/* This for loop constitutes set of variables present in module and compares them with the variables present in dependent set of modules */

{

For each module Mj // where j=1 to n and i≠j

{

Find CDij, where CDij is the set of common variables between the dependent modules

/* CDij is calculated by taking the intersection of the variables from the dependent set of modules */

Find Calculated factor (CFij) =

$$\frac{\textit{common variables between dependent modules}}{\textit{Total variables in dependent modules}}$$

}

}

For each module Mi // where i=1 to n

{

/* This for loop accounts for set of variables along with the variables that are responsible in producing defects and compares them with the variables present in dependent set of modules */

For every dependent module Mj //where i≠j

{

Find DVij, where DVij is the set of defect producing variables participating in the dependent modules and are responsible for producing defects.

/* DVij is calculated by taking the intersection of the variables that are participating in the dependent modules and are responsible for producing defects */

Calculate percentage Defect propagation factor (DFij)=

$$\frac{\textit{defect producing variables participating between dependent modules}}{\textit{common variables between dependent modules}}$$

}

}

For each module Mi // where i= 1 to n

{

For every dependent module Mj // where j=1 to n i≠j

{

If (DFij) >= (CFij)

/*Where (DFij) is calculated above as

$$\frac{\textit{defect producing variables participating between dependent modules}}{\textit{common variables between dependent modules}}$$

And CFij is calculated above as

$$\frac{\textit{common variables between dependent modules}}{\textit{Total variables in dependent modules}}$$ */

Then module is fault prone

Else

Module is not fault prone

}

}

From the proposed algorithm if the Calculated Factor is less than the percentage defect propagation factor then there is no defect between the modules and the coupling between the modules is less. But if the Calculated Factor is greater than the

percentage defect propagation factor then the probability of the module to be fault prone increases and testing team have to focus more on checking that particular fault prone modules, so that the defect should not propagate in the whole system. This approach is more efficient in cost and time for the testing team and for the software industry.

7.2. Example

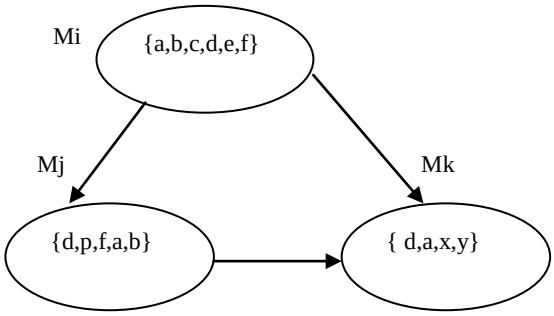

Fig 3 : Set of Dependent Modules

.

Some abbreviations that are use in the paper for calculating tasks:

Calculated Factor (CF) between the set of dependent modules =

$$\frac{\textit{common variables between dependent modules}}{\textit{Total variables in dependent modules}}$$

Defect Propagation factor (DF) between the set of dependent modules =

$$\frac{\textit{defect producing variables participating between dependent modules}}{\textit{common variables between dependent modules}}$$

For example, Consider three modules as per the given figure and try to find out the probability of fault prone module.

Let i={a,b,c,d,e,f }and j= {d,p,f,a,b} are set of variables in Module i and Module j

CDij= {a,d, f, b} are set of common variables between the dependent set of modules i and j

DVij= {a,b},where DVij is the set of defect producing variables participating in the dependent modules.

Defect propagation factor (DFij) =2/4 or 50%(or in other words it can be said that if module i is producing defect than statistically it could be said that there is approximate 50% probability , module j would also produce defect)

CFij=4/7 =0.57 or 57% is the calculated factor in module i

Let j={d,p,f,a,b} k={d ,a,x,y}be the set of variables in modules j and k.

CDjk= {d, a} be the set of common variables between the dependent set of modules i and j

DVjk= {d, a} is the set of defect producing variables participating in the dependent modules.

Defect propagation factor (DFjk) =2/2 or 100 %(or in other words it can be said that if module j is producing defect than statistically there is approximate 100% probability, module k would also produce defect)

CFjk=2/7 =.28 or 28% is the calculated factor in module j

Let i={ a,b,c,d,e,f } k={ d ,a,x,y}are set of variables in modules i and k

CDik= {a, d} be the set of common variables between the dependent set of modules i and k

DVik={a} is the set of defect producing variables participating in the dependent modules

Defect propagating factor (DFik) =1/2 or 50%(or in other words it can be said that if module i is producing defect than statistically there is approximate 50% probability , module k would also produce defect)

CFik=2/8 =.25 or 25% is calculated factor in module k.

It is clear from the above example that CFjk=28%but DFjk=100%, it means the probability that module j is fault prone increases, as module j contains all the variables that are producing defects which are defined in module i.So the testing team will now focus more on module j, and this lead to an increased efficiency of the testing team which is a prime concern of a software industry. Greater the Percentage defect propagation Greater will be the probability of the module to be fault prone and this is directly proportional to increased coupling. As for a module or a system the coupling should be as low as possible which a prime concern in software engineering field. On detecting the fault prone module the testing team will focus more on that particular module so that in near future the defect propagation from that module should be minimized.

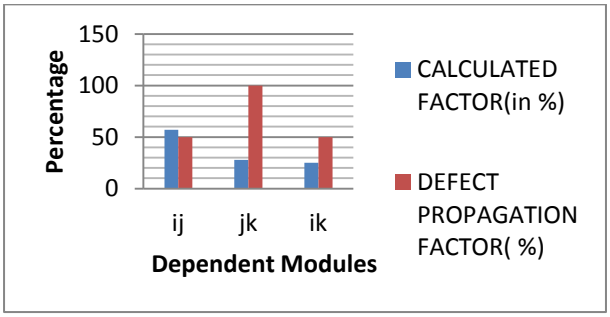

Fig 4: Plot showing variation of defect propagating factor w.r.t calculated factor between dependent modules

7. RESULTS & COMPARISION

The results presented below demonstrate the comparison of Calculated Factor with respect to the defect propagating factor between the dependent modules. Higher the Defect propagation factor of the dependent module with respect to the Calculated Factor shows that the probability of the module to be fault prone is high. This high fault prone probability is directly proportional to the coupling.

Calculating the failure-correlation based on the existing algorithm in the given example and comparing the defect propagation factor of the proposed algorithm with the failure-correlation of the existing algorithm, it can be seen that the Defect propagation factor calculated by proposed algorithm is greater than the failure-correlation as calculated based on the existing algorithm. So the proposed algorithm gives higher probability to detect the module to be fault prone as compared to existing algorithm.

Table 1: Results showing comparison between existing approach and proposed approach in terms of Defect Propagating Factor and Failure-Correlation

Dependent modules	Defect Propagation Factor(%)	Calculated factor(%)	Failure-Correlation
(i,j)	50	57	53.45
(i,k)	50	25	26.73
(j,k)	100	28	53.45

As seen from the above table, the Defect Propagation factor of dependent modules (i,k) and(j,k) is statistically much higher than the failure correlation factor of existing algorithm.So,the probability that modules (i,k) and (j,k) are more fault prone is high and also the coupling between these dependent modules is also high.

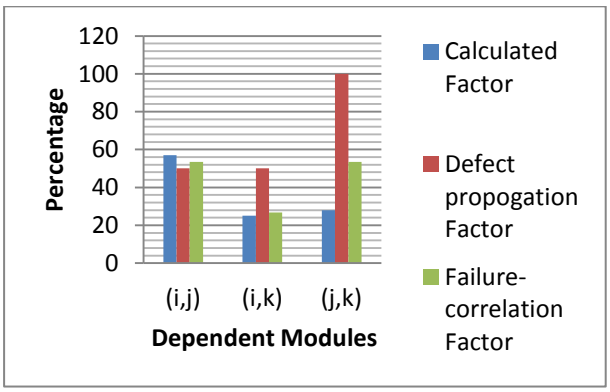

Fig 5: Comparison of Various Factors of Existing and Proposed Algorithm

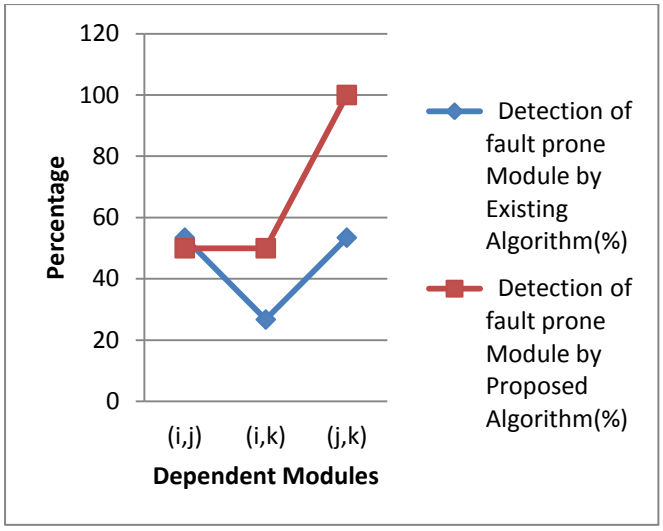

Fig 6: Comparison of Both Approaches in Predicting Fault Prone Module

The table shown below gives the relationship that how the fault prone dependent modules are directly related to coupling. It is shown in the result that the probability of the dependent modules (j,k) and (i,k) to fault prone is high and due to which the coupling between them is also high.

Table 2: Results of Proposed approach showing relation between fault prone modules and coupling

Dependent modules	Defect Propagation Factor(%)	Calculated factor(%)	Probability of fault prone	Coupling
(i,j)	50	57	no	low
(i,k)	50	25	yes	high
(j,k)	100	28	yes	high

8. CONCLUSION

The focus is on how knowledge discovery can be applied on the software modules by analyzing and predicting the defects present in them. For that an algorithmic approach have been proposed for directly coupled interlinked software modules, which gives a probability to developers as well as testing team members, to concentrate more on the defect prone modules, so as to minimize the defects in software modules. This will led to improve the efficiency as well as saves a lot of time of the testing team in the software industry. Later on, this work can also be extended for indirect coupled modules.

9. REFERENCES

[1] Fang Deng, James A. Jones," Weighted System Dependence Graph", 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation"

[2]. Tan, Xi Sch. of Comput. Sci., Fudan Univ., Shanghai, China Peng, Xin, Pan, Sen, Zhao, Wenyon," Assessing Software Quality by Program Clustering and Defect Prediction", Reverse Engineering (WCRE), 2011 18th Working Conference, pp. 244 – 248, Oct. 2011, ISSN : 1095-1350

[3] Vinay Singh and Vandana Bhattacherjee," Detection of Indirect Coupling Using Chaining Method and Its Impact on Software Quality", International Journal of Research and Reviews in Information Sciences (IJRRVol. 1, No. 4, December 2011, ISSN: 2046-6439

[4]. Gonzalez-Sanchez, Alberto Software Technol. Dept., Delft Univ. of Technol., Delft, Netherlands Abreu, Rui, Gross, Hans-Gerhard, Van Gemund, Arjan J C," Prioritizing tests for fault localization through ambiguity group reduction", Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference, pp**.** 83 – 92, 6-10 Nov. 2011, ISSN : 1938-4300

[5]. Jalbert, Kevin Software Quality Res. Group, Univ. of Ontario Inst. of Technol., Oshawa, ON, Canada Bradbury, Jeremy S.," Using clone detection to identify bugs in concurrent software", Software Maintenance (ICSM), 2010 IEEE International Conference, pp**.** 1 – 5, 12-18 Sept. 2010, ISSN : 1063-6773

[6]. Nugroho, Ariadi LIACS, Leiden Univ., Leiden, Netherlands Chaudron, Michel R V,Arisholm, Erik," Assessing UML design metrics for predicting fault-prone classes in a Java system:, Mining Software Repositories (MSR), 2010 7th IEEE Working Conference, pp. 21 – 30, 2-3 May 2010, Print ISBN: 978-1-4244-6802-7

[7] W. E. Wong, V. Debroy and B. Choi, "A Family of Code Coverage-based Heuristics for Effective Fault Localization," Journal of Systems and Software, 83(2):188-208, February, 2010

[8]. Chen, Yuan Changchun Inst. of Opt., Fine Mech. & Phys., Chinese Acad. of Sci., Changchun, China Shen, Xiang-heng, Du, Peng,Ge, Bing,"Research on software defect prediction based on data mining", Vol. 1,Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference,563 – 567, pp**.** 26-28 Feb. 2010, E-ISBN : 978-1-4244-5586-7

[9]. Liu Yanbin Ordnance Eng. Coll., Shijiazhuang, China,Zhu Xiaodong,Sun Zhiming,Wang Yigang,Ye Fei," Dual-Slices Algorithm for Software Fault Localization", Computational Intelligence and Software Engineering, 2009. CiSE 2009. International Conference, pp**.** 1 – 4, 11-13 Dec. 2009, Print ISBN: 978-1-4244-4507-3

[10] P.-N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005

[11] F. Tip, "A survey of program slicing techniques, "Journal of Programming Languages,3(3):121–189, 1995

[12] M. Weiser, "Program slicing," IEEE Transactions on Software Engineering, SE-10(4):352-357, July 1984

[13] J.-D. Choi, B. P. Miller, and R. H. B. Netzer, "Techniques for debugging parallel programs with flowback analysis," ACM Trans. Program. Lang.Syst., vol. 13, pp. 491–530, October 1991.

[14] S. Bates and S. Horwitz, "Incremental program testing using program dependence graphs," in Symposium on Principles of Programming Languages. New York, NY, USA: ACM, 1993, pp. 384–396.

[15] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Trans. Program. Lang. Syst., vol. 12, pp.26–60, January 1990

[16] K. J. Ottenstein and L. M. Ottenstein, "The program dependence graph in a software development environment," SIGPLAN Not., vol. 19, no. 5,pp. 177–184, 1984.

[17] Briand,L.C., Daly,J.W., & Wust,J.K.,"A Unified framework for coupling measurement in object oriented system". IEEE Transact Software Engineering, (25(1): pp. 91-121, January/February 1999

[18] Yourdon. & Constantine, L.L," Structured Design: Fundamental of a discipline of computer program and system design prentice hall", 1979

[19] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy ofspectrum-based fault localization," in Testing: Academic and Industrial Conference Practice and Research Techniques, 2007, pp. 89–98.