A Framework for Performing Mutation Analysis and Deviants

Manoj Kumar, Mohammad Hussain

Abstract*— The development of framework for safety critical area what happens, when some part of a system deviates from the intentions of designer is a critical research issue. When we apply, HAZOP technique using UML, then, we check the object-oriented design with a fault-free analysis and design. By mutation analysis and HAZOP, we find a better optimum result. The mutation method is a fault-based testing strategy that measures the quality/adequacy of testing by examining whether the test set (test input data) used in testing can reveal certain types of faults. This paper describes the UML-HAZOP technique with mutation based operator or analysis. Using this, we find more and more optimum result and solution, when we design our system with UML.

***Index Terms*—Mutation Analysis, Mutation Testing, UML-HAZOP, Object-Oriented.**

I. INTRODUCTION

In the current scenario, developing software is a lucrative business globally. Often, the development of object oriented models using UML notation is attaining more popularity in the market of software engineering. But along this one major problem is also concerning with is presence of errors that can cause some trouble to the user in future. Correction of such kinds of defects/errors is not a big deal. We can overcome from these problems, but one truth is also seen that is recognition of errors and their correction demanding enough time and money. Nowadays many companies hiring such professionals, those, who are experts in troubleshooting and they are paid high for this only. If these defects are not found out at the right phase of development and delivered to the client with some defects, resulting the huge loss to the supplier and his/her brand is also overshadowed.

The mutation method [DeMillo78] is a fault-based testing strategy that measures the quality/adequacy of testing by examining, whether the test set (test input data) used in testing can reveal certain types of faults. Unlike, other fault-based strategies that directly inject artificial faults into the program, the mutation method generate simple syntactic deviations (mutants) of the original program, representing 'typical' programming errors. The mutation techniques are used to examine the adequacy of test data for object-oriented programs. In object-oriented systems, mutation testing should also consider the relationships between components even though it is presently aimed at testing a single method or a class. [1].

One problem in the design of testing experiment is that real programs of appropriate size with real faults are hard to find, and hard to prepare appropriately (for instance, by preparing correct and faulty versions). Even, when actual programs with actual faults are available, often these faults are not numerous enough to allow the experimental results to achieve statistical significance. Many researchers therefore have taken the approach of introducing faults into correct programs to produce faulty versions. The main potential advantage of mutant generation is that the mutation operators can be described precisely and thus provide a well-defined, fault-seeding process. This helps researchers replicate others' experiments, a necessary condition for good experimental science. While hand-introduced faults can be argued to be more realistic, ultimately it is a subjective judgment whether a given fault is realistic or not. Another important advantage of mutant generation is that a potentially large number of mutants can be generated, increasing the statistical significance of results obtained [2].

The aim of presenting this paper is to noticing the defects that comes during the early phase of software development and making sure that the fault free software is delivered to the user. The purpose of choosing this topic like defects introduced in the object oriented models expressing UML notation, because it is triggered/interpreted by man itself.

In this paper, our approach uses the framework for performing mutation analysis and deviants with UML-HAZOP technique for evaluating the object-oriented model. This approach uses HAZOP (hazard and operability studies) technique – a technique used in the safety critical area to systematically investigate and record what happens when some part of a system deviates from the intentions of the designer. When we apply HAZOP technique using UML then we checks the object-oriented design with a fault-free analysis and design (see figure 1).

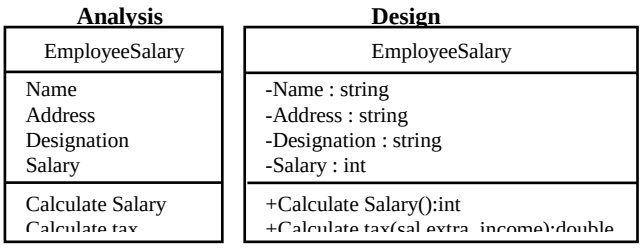

Fig. 1: Analysis & Design Versions of a Class

By this technique, we find a better analysis and design for real world application. We analyze the design in a better way, and applying guidewords to them, we identify valid deviations and then this design to derive by mutation operators that would give rise to them.

Manuscript received August 22, 2011.

Manoj Kumar, Research Scholar, UPRTOU, Allahabad, India, 9956010822, (e-mail: iisemanoj@gmail.com).

Dr. Mohammad Husain, Director, AIET, Lucknow, India, 9415459591, (e-mail: mohd.husain90@gmail.com).

If chased thoroughly, this technique will cover every design construct and feature. In section II, a review of mutation with object-oriented techniques is presented. In section III, show the HAZOP technique. In section IV, we identify the attribute list of object- oriented constructs. In section V, we evaluate the mutation operator using UML-HAZOP technique. Finally, section VI presents conclusion.

II. RELATED WORK

C Roger T. Alexander [3] has proposed a technique for performing mutation analysis on object-oriented programs by injecting faults into objects. This technique makes mutation work for OO software. He showed that reusable libraries of mutation components can effectively inject plausible faults into objects that instantiate items from common Java libraries as well as user defined classes. He designed an object mutation engine that implements his technique. Chanchal K. Roy and James R. Cordy [4] propose a new approach for evaluating clone detecting tools in a controlled way by borrowing an established technique from the testing community- mutation based analysis. He has not yet completed the implementation of the framework; such a framework can provide concrete and accurate comparative results for different tools in finding intentionally created code clones. In this proposed framework, it is not practical to work with large scale code bases. Sunwoo Kim et al.[1] have extended the traditional mutation method by proposing a set of mutation operators that are intended to represent plausible flaws related to the unique features in object-oriented (Java) programs. The Class Mutation technique can be used in itself as a form of object-oriented directed selective mutation testing or it can be integrated with the conventional mutation systems. P. Chevalley et al. [5] presents the first prototype GUI-based tool supporting mutation analysis of Java programs. The tool implements 26 mutation operators (including 20 object-oriented specific operators) targeting various types of plausible faults in a Java program. Two factors have motivated this tool: first, mutation analysis is a powerful and computationally expensive fault-based technique that cannot be considered without the aid of an efficient tool taking an active part in the automation of the technique; second, the Java language integrates object -oriented features that can be the basis for new types of faults non-existent in procedural languages. Sunwoo Kim et al. [6] propose the use of a safety technique known as HAZOP (Hazard and Operability Studies) to rigorously generate mutation operators for Java. A set of Java mutation operators is proposed by applying HAZOP to the Java syntax definition and is compared to the operator sets of current mutation systems. Janusz GÓRSKI et al. [7] present a method supporting detection of defects in UML based software documentation. This method named UML-HAZOP is the adoption of HAZOP (Hazard and Operability Studies) – a technique widely applied to safety-related systems, and concentrates on analyzing "flows" between system's components in order to detect anomalies related to these flows. He describes the method and the results of some experiments related to its application to two real systems: a billing system of a telephone exchange and a management support system. Janusz Górski, Aleksander Jarzębowicz[8] introduces the UMLHAZOP and presents results of its validation through a series of case studies and controlled experiments. He introduces a new technique of software inspection. It distinguish feature is that it focuses on the UML models that are produced early during software development. Another distinguishing feature is that the method is based on the checklist that can be generated is a systematic way using a set of HAZOP guidewords. This process of checklists generation has been automated if the UML models are important from a common CASE tool.

III. UML- HAZOP METHOD

The HAZOP technique was initially developed to analyze chemical process systems, but has later been extended to other types of complex systems including, as examples, transportation systems and software systems. The HAZOP process is just one of a large number of different techniques available to the safety professional for analyzing systems to identify and prevent hazards. It has the further advantage that it also identifies and helps to prevent operational problems. A Hazard and Operability (HAZOP) method is a structured and systematic examination of a planned or existing process or operation in order to identify and evaluate problems that may represent risks to personnel or equipment, or prevent efficient operation. The HAZOP study should preferably be carried out as early in the design phase as possible - to have influence on the design. As a compromise, the HAZOP is usually carried out as a final check, when the detailed design has been completed. A HAZOP study may also be conducted on an existing facility to identify modifications that should be implemented to reduce risk and operability problems [9].

We use the set of guidewords proposed in the HAZOP standard [10]. It is shown in Table 1.

Table 1: Generic HAZOP Guidewords

GUIDEWORDS	GENERIC INTERPRETATION
NO	The complete negation of the design intention. No part of the intention is achieved and nothing else happens.
MORE	A quantitative increase.
LESS	A quantitative decrease.
AS WELL AS	All the design intention is achieved together with additions.
PART OF	Only some of the design intention is achieved.
REVERSE	The logical opposite of the intention is achieved.
OTHER THAN	Complete substitution, where no part of the original intention is achieved but something quite different happens.
EARLY	Something happens earlier than expected relative to clock time.
LATE	Something happens later than expected relative to clock time.
BEFORE	Something happens before it is expected, relating to order or sequence.

AFTER	Something happens after it is expected, relating to order or sequence.

UML-HAZOP [7] is an adoption of HAZOP which focuses on design defects that are present in UML models. It is mainly use in UML diagram for defects detection and structured review method for UML diagram guided by keywords (NO, MORE, LESS, PART OF etc). It is checklist for UML diagram (Table2).This technique is useful for analysis. The method can aim at hazard analysis, in which case, it looks into dangerous consequences of the considered deviations, or defect analyses [11]. This technique is suitable for where the analysis is required and it also use in quality assurance process.

Table 2: Guide words for UML-HAZOP

GUIDEWORDS	GENERIC INTERPRETATION
NO	No part of the intention is achieved and nothing else happens. No name though it should be named.
MORE	A quantitative increase, The multiplicity & no. of classes are too high.
LESS	A quantitative increase, The multiplicity & no. of classes are too low.
AS_WELL_AS	Specific design should not take place, but it achieved with additional results.
PART_OF	Only some of the intention is achieved, not yet finished.
REVERSE	Flow of information in wrong direction, reading the association name between the classes.
OTHER_THAN	Wrong type of relationship, name should be changed or removed.
EARLY	Some design make earlier than the expected design.
LATE	Some design make late than the expected design.
BEFORE	Some design make before it is expected, relating to order or sequence.
AFTER	Some design make after it is expected, relating to order or sequence.

To adopt HAZOP to UML notation, it is necessary to define which element of particular UML diagrams are considered as HAZOP "connection" and "attribute" and which type of anomaly is suggested by applying a particular HAZOP guidewords to a particular attribute. As the result, we obtain checklist that list all possible anomalies of a considered element of the model [7].

We take an idea from adopted HAZOP:

[a] In the beginning of software development life cycle, we analyze UML model.

[b] To analyze, if the defected anomalies can have negative consequences downstream the development process. The defect of particular interest were design defects (wrong mapping of real world concepts into a model), violation of "good practice" (for instances too much responsibility assigned to a class, too many level of inheritance) or violation of some system attributes, like performance or security [8].

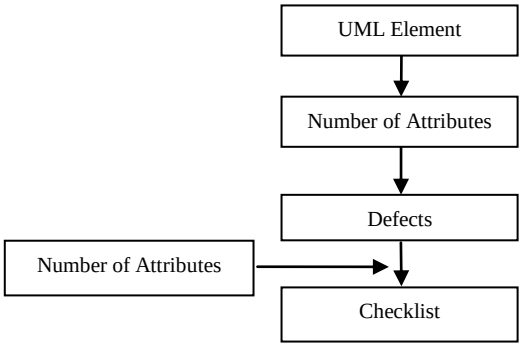

Fig. 2: UML-HAZOP Checklist

In figure 2[11], show the checklist of UML-HAZOP technique for check the defects. Each application of a HAZOP guideword to an UML entity or attributes results in a suggestion of a model anomaly. Such candidate anomalies are subjected to a preliminary analysis with respect to their credibility and are eventually inserted in the checklists. Some examples are given in table 2.

IV. ATTRIBUTE OF OBJECT- ORIENTED CONSTRUCTS

Table3 shows the list of attributes identified for UML constructs. In the below table 3, we are generally identify the attribute of UML and we take a these construct in the next section for mutation.

Table 3: General Attributes of UML Constraints

CONSTRUCTS	ATTRIBUTES
Class	• Class names • Member function • Member attribute • Visibility
Association	• Association name • Multiplicity • Role name • Visibility
Generalization	• Parent object • Child object • Is_aggregation
Events	• Event name • Caused_by • Related action • Effected member
States	• State name • Caused_by(Source event) • isFinal(Boolean type)

V. OUR METHODOLOGY

In figure 3, we give a real life application and make a design in UML form and then we implement in UML-HAZOP technique for safety checking and then mutate the operator. After mutation, we find some fault and we killed those fault after that we find fault free design. This methodology, we are implementing for this paper.

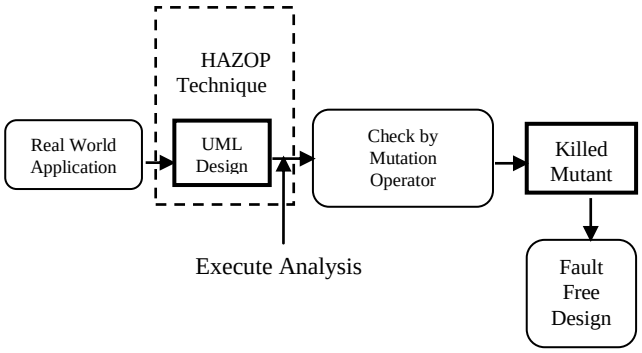

Fig. 3: Mutation Analysis Based Framework with UML-HAZOP Technique

We identify some of mutation operator for UML and we do not claim that these set of operator is complete as a mutation operator.

a) Class based mutation operator
b) Association based mutation operator
c) Generalization based mutation operator
d) Event based mutation operator
e) State based mutation operator

A. CLASS

A class describes a set of objects with similar structure, behavior and relationships. Classes are defined by a set of attributes and operations in a class diagram.

Syntax of Class: The class is shown as a rectangle box

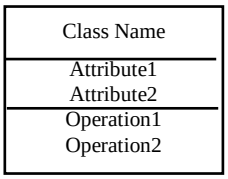

Fig. 4: Structure of Class

An attribute describes a range of values that instances of the class may hold. It is defined by a name and a type. Additionally, an attribute can have properties like visibility (to other classes), multiplicity, an initial value and a property-string that indicates property values.

Visibility name [multiplicity] : type-expression = initial-value {property-string}

Operations are specified by a name and a optional list of arguments.

Visibility name (eparameterlist) : return-type-expression {property-string}

Applying these UML-HAZOP's guidewords to the attribute 'class' produces the following deviants (table 4). An object is an instance of a class. In UML an object is represented by a rectangle with one or more compartments (up to four compartments). The top compartment shows the name of the object and the name of the class. The other compartments can be suppressed.

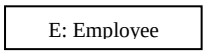

Fig. 5: An object in UML

An object represents a particular instance of a class and the same notation is used in collaboration diagrams to represent roles because roles have instance-like characteristics.

An object represents a particular instance of a class and the same notation is used in collaboration diagrams to represent roles because roles have instance-like characteristics.

Table 4: UML-HAZOP Guidewords for the Attribute of Class

Attribute: Class	**Guideword: NO**	**Cause:** Attribute of instance does not match the class member.
		Consequences: An object is not a member of an expected class.
	Guideword: MORE	**Cause:** Can be use more than one class.
		Consequences: A class has more/fewer instance than expected.
	Guideword: PART OF	**Cause:** Some instance have extra attribute other than class, some matching attribute but not all/some missing.
		Consequences: Some of the class constraints are true, other are not.
	Guideword: OTHER THAN	**Cause:** A class is defined as friend of another class and its objects can access private and protected data members of that class.
		Consequences: An object is a member of an unintended class.
	Guideword: LESS	**Cause:** We use only one class.
		Consequences: A class has less than two instance or equal to one.

Mutation Operator for Class

The clearness of operations those come under class is changeable. User would not face problem, if we are changing the behavior of private in a public operator. In the condition if public operator is going to be private then there is undesirable changes showing in the behavior of operators. If same operation name is using in main class and its subclass and that same operation is calling then function of the main class is calling first and in that condition there is no requirement of calling the function of the subclass and this operator being remain in the subclass [12].

B. Association

An association defines a relationship between two or more classes. Binary associations are relationships between exactly two classes and a n-ary association is an association between three or more classes.

Syntax:

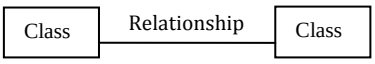

Fig. 6: Class1 Is Associated With Class2

The following table shows the deviants in the attributes 'scope' and 'accessibility' of Name construct. The mutation operators "replacing SimpleName to QualifiedName (or vice versa)" and "replacing an access modifier with alternatives" are derived from the deviants.

Table 5: UML-HAZOP Guidewords for the Attribute of Association

Attribute: Association	**Guideword: AS WELL AS**	**Cause:** When the classes are not actually an associated. In That condition, association will be false and should be removed from the diagram.
		Consequences: It is not necessary that we take in every time more or less than one class.
	Guideword: PART OF	**Cause:** If association is true and some extra relationship or relationship should be introduced between the actual classes.
		Consequences: Check the relationship that it is generalization.
	Guideword: OTHER THAN	**Cause:** Relationship between associations should be correct.
		Consequences: If a relationship is false and a relationship of different type. This condition rise, when the generalization should replace the association.

Mutation Operator for Association

Association has several attributes: association name, multiplicity, role name, visibility. When we change the behavior between classes, then there is no right association in that case, we mutate between the classes. By changing the value, we can mutate the association. An association end multiplicity may be changed. "1…*" can be converted to "*". If the operator results in a weaker constraint, the mutant may be equivalent. When we change a private operation into a public one , or relax the multiplicity constraints, The mutant may be equivalent, if we find the mutation operator's result in a weaker restraints. The importance of the default remains in the model even if a weaker constraint generates an equivalent mutant. This happens when we change a private operation into a public one, or relax the multiplicity constraints. However, even if a weaker constraint generates an equivalent mutant, the fault is still an important one in the model [12].

C. Generalization

A generalization is shown as a line with a hollow triangle as an arrowhead between the symbols representing the involved classifiers.

The arrowhead points to the symbol representing the general classifier.

Syntax:

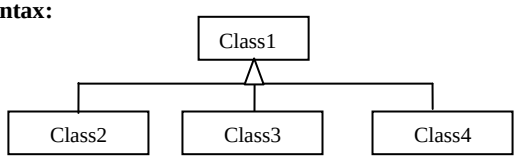

Fig. 7: Class2, Class3 and Class4 are generalized by Class1

Table 6: UML-HAZOP Guidewords for the Attribute of Generalization

Attribute: Generalization	**Guideword: AS WELL AS**	**Cause:** Two classes are associated but are not in a hierarchy.
		Consequences: Considered classes are not really related with Generalization relationship. The relationship is wrong and should be removed from the diagram.
	Guideword: MORE	**Cause:** Classes having all attribute similar, are made child class to same parent class.
		Consequences: Some additional classes are marked as subclasses; they shouldn't take part in this relationship.
	Guideword: LESS	**Cause:** Some classes (present on the diagram or not) that should take part in this relationship are not marked as subclasses.
		Consequences: It is not necessary that every time more than one class is generalized or it is not a concrete class.
	Guideword: OTHER THAN	**Cause:** Objects of two classes are not really related through hierarchy. They are related in some other way.
		Consequences: Wrong type of relationship. A relationship of another type should be defined e.g. association instead of generalization.

Mutation Operator for Generalization

Wrong Generalized Tree: There is no guarantee that when we decompose a problem then we shall find a better result. If decomposition is less specific and unfocused way then we will not get a better result. If we generalized a parent class to child class and child class to grandchild class and we are not getting a suitable result then we mutate the grant child class to all child class.

If we get better result then we stop this mutation otherwise, and we don't get better result then we again mutate to grant child class to parent class and again if find a better result then we stop this exercise, so we mutant between all classes.

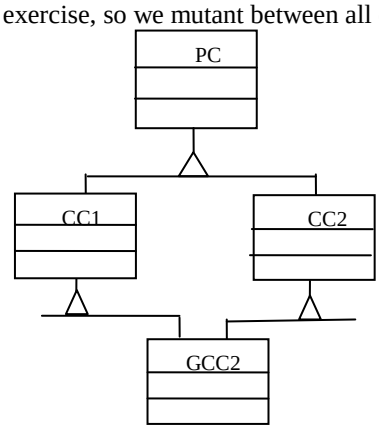

Fig. 8: Wrong Generalization Tree

D. Events

Every event is a unique occurrence.

Syntax:

```
event(arg1,arg2,arg3)
event(arg1,arg2)
Digit dialed(arg1)
```

Fig. 9: Event classes and attributes

Table 7: UML-HAZOP Guidewords for the Attribute of Event

Attribute: Event	**Guideword: NO**	**Cause:** Object of derived class calls the base class method and the method of same name is also in derived class (method of parent class is overridden by child class function).
		Consequences: Event not received by control system, either it occurs but is not transmitted to the controller because of sensor or other failure, or it does not occur even though expected.
	Guideword: AS WELL AS	**Cause:** Another event is detected by the control system as well s the intended event.
		Consequences: Two events are similar.
	Guideword: OTHER THAN	**Cause:** An unexpected event is detected instead of the expected event.
		Consequences: A runtime error is generated or an anomalous situation is occurred. Two events of similar type exist.

Mutation Operator for Event

Two events that are casually unrelated are said to be concurrent, they have effect on each other then we change the state model. If we find a wrong event, then we replace the event with another event from the state model in same changeover to this operator's result. Another operator is missing event, in that case, we take this operator results in the remotion of just one event from the original state model [13]. We generate mutation operator our corresponding our mutation list to each event by removing that event from the model.

E. States

A state is drawn as a box with rounded corners. Each state models a set of possible object values that have similar behavior - but possibly different attribute values. A state is in a different box if objects in these states behave differently.

Syntax:

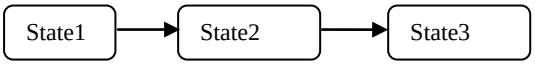

Fig. 10: State diagram

Table 8: UML-HAZOP Guidewords for the Attribute of States

Attribute: State	**Guideword: NO**	**Cause:** Object use the values or access the methods of the class, to which it is not related in a meaningful way.
		Consequences Object not in expected state
	Guideword: OTHER THAN	**Cause:** Object tries to copy the state of other object and both objects are not associated to each other.
		Consequences: Object in an unexpected state

Mutation Operator for States

Incorrect state operator can be exchanged from on state to another thus result can also be modeled in the right state. In the case of missing operator state, one state can be converted into original or right state. If state is not starting from the right one, in that case starting state is impulse. Next operator is incorrect action. If action is not correct in the state model then mutating one action into another.

Another operator is change guard condition, this operator is not resulting the right guard expression, in that case expression of corresponding guard condition is to be set in the state [13].

VI. CONCLUSION & FUTURE SCOPE

Software analysis and designing is one of the most crucial tasks in the software development process. If we are not giving more emphasis on this step, then we have problems related to money as well as time constraints. Developing of object oriented models using UML notation is attaining more popularity in the market of software engineering. The aim of presenting this paper is to notice the defect that comes during the early phase of software development and making sure that the fault free software is delivered to the user.

The purpose of choosing this topic like, defects introduced in the object oriented models, expressing UML notation, because it is triggered/interpreted by man itself.

In this paper, our approach uses the framework for performing mutation analysis and deviants with UML-HAZOP technique for evaluating the object-oriented model. This approach uses HAZOP (hazard and operability studies) technique – a technique used in the safety critical area to systematically investigate and record what happens when some part of a system deviates from the intentions of the designer. When we apply HAZOP technique using UML then we checks the object-oriented design with a fault-free analysis and design.

By the application of this technique, we find a better analysis and design for real world applications. We analyze the design in a better way and applying guidewords to them, we identify valid deviations and then this design to derive by mutation operators that would give rise to them. If chased thoroughly, this technique will cover every design construct and feature. In future, by this technique, we can solve the problem when we developing our application in object-oriented form.

REFERENCES

1. S. Kim, J. A. Clark & J. A. McDermid, "Class Mutation: Mutation Testing for Object - Oriented Programs", in Net. Object Days Conference on Object - Oriented Software Systems, 2000.
2. J. H. Andrews, L .C. Briand and Y. Labiche, "Is Mutation An Appropriate Tool for Testing Experiments", Proceeding 27th International Conference on Software Engineering, St Louis, USA, 2005, pp. 402-411.
3. R. T. Alexander, J. M. Bieman, S. Ghosh, Bixia Ji, "Mutation of Java Objects", To appear in Proc. IEEE International Symposium Software Reliability Engineering (ISSRE), 2002.
4. C. K. Roy, J. R. Cordy, "Towards a Mutation -Based Automatic Framework for Evaluating Code Clone Detection Tools", ACM International Conference Proc. Series, Volume: 273, Publisher: ACM Press, pp 137-140, 2008.
5. P. Chevalley, P. Th´evenod-Fosse, "A Mutation Analysis Tool for Java Programs", International Journal Software Tools Technology Transfer , 2003.
6. [6] S. Kim, J. A. Clark, and J. A. McDermid, "The Rigorous Generation of Java Mutation Operators Using HAZOP", In Proceedings of the 12th International Conference on Software and Systems Engineering and their Applications (ICSSEA' 99), Paris, France, Dec-1999.
7. J. Górski, A. Jarzębowicz, "Detecting Defects in Object - Oriented Diagrams Using UML - HAZOP", Foundations of Computing and Decision Sciences, Vol. 27, No. 4, 2002.
8. J. Górski , A. Jarzębowicz, "Development & Validation of a HAZOP Based Inspection of UML Models", 3rd World Congress for Software Quality 26-30 Sep- 2005,
9. M. Rausand, "HAZOP Hazard and Operability Study", System Reliability Theory (2nd ed), Wiley, pp. 1- 44. 2004.
10. Ministry of Defense, "HAZOP Studies on Systems Containing Programmable Electronics", Defense Standard 00-58, Parts 1 and 2, Issue 2, May 2000.
11. A Jarzębowicz and J Górski, "Experimental Comparison of UML-HAZOP Inspection & Non – Structured Review", Found. of Computing and Decision Sciences, Vol. 30. No. 1, pp. 29-38, 2005.
12. Dinh -Trong, S. Ghosh, R. France, B. Baudry, and F. Fleurey. "A Taxonomy of Faults for UML Designs", In Proceedings of MoDeVa'05 (Model Design and Validation Workshop associated to MoDELS'05), Montego Bay, Jamaica, October 2005.
13. S. B. A. Punuganti, P. k. Pattanaik, S. Prasad, R. Mall, "Model-Based Mutation Testing of Object-Oriented Programs", Proceedings of 2nd International Conference on IT & Business Intelligence (ITBI-10), Nagpur, INDIA, November12–14, 2010.

AUTHORS PROFILE

Manoj Kumar: He is a Research Scholar of Computer Science & Engineering, Uttar Pradesh Rajarshi Tandon Open University, Allahabad, India. He got his M.Tech. Degree in Computer Science & Engineering and is also an MCA degree holder. He posses more than 09 years of experience in teaching and 02 years of software development experience. Currently, he is actively engaged in the research work in the field of Software Engineering. He has also published books and several research papers.

Prof. (Dr.) Mohd. Husain: Presently working as Director, AZAD Institute of Engineering and Technology, Lucknow, India. He Received Ph.D. Degree from Integral University, Lucknow in 2008 and Master Degree (M.Tech.) from UP Technical University, Lucknow. He has about 21 years of experience in IT & Academics and 07 years research experience. He has published more than 130 International and National publications.