Fault Classification and Automated Test Case Generation by Using Mutation Testing

Milind Kale[1], Vikas Mapari[2]

[1]Department of Computer, D.Y.Patil College of Engineering, Ambi, Pune, India

[2]Professor, Department of Computer Science, D.Y.Patil College of Engineering, Ambi, Pune, India

Abstract: ***Software testing is that the necessary section of software development method. But, this section may be simply missed by package developers due to their restricted time to complete the project. Since, package developers end their software nearer to the delivery time; they don't get enough time to check their program by making effective test cases. One of the major difficulties in package checking is that the generation of test cases that satisfy the given adequacy criterion what is more, creating manual check cases may be a tedious work for package developers within the nil rush hours. A replacement approach that generates check cases will facilitate the package developers to form test cases from package speciation's in early stage of package development (before coding) and furthermore as from program execution traces from once package development done (after coding). Heuristic techniques may be applied for making quality check cases. Mutation testing may be a technique for testing package units that has great potential for raising the standard of testing, and to assure the high dependableness of package. during this paper, a mutation testing based check cases generation technique has been planned to generate check cases from program execution trace, in order that the check cases may be generated once cryptography performed. The paper details concerning the mutation checking implementation to get test cases.***

Keywords: *Fault Classification, fault-revealing, Mutation testing.*

1. Introduction

In modern computer systems, system event logs have invariably been the rst supply for checking the system standing. As computer systems become more advanced, like cloud computing systems, the interaction among code and hardware is progressively often. These elements can generate monumental log info, as well as running reports and fault information. The huge information could be a nice challenge for analysis with manual methodology. We tend to implement a log management and analysis system, which may assist system directors to know the period standing of the complete system, classify logs into totally different fault sorts, and confirm the foundation reason behind the faults. Additionally, we tend to improve the prevailing fault correlation analysis methodology supported the results of system log classication. We tend to apply the log management and analysis system to cloud computing setting for analysis. The results show that our system will classify fault logs effectively and mechanically. By exploitation the planned system, directors can easily detect the root cause of faults. Previous technique consists of two steps Training, Training is a preprocessing that extracts properties of programs for which errors are known a priori, converts these into a form amenable to machine learning, and applies machine learning to form a model of fault-revealing properties. Classication step is the user supplying the model with properties of new code to select the fault revealing properties, and using those properties to locate latent errors in the new code. The training step of the technique requires programs with faults and versions of the same programs with those faults removed. The programs with faults removed need not be error-free, and also the ones employed in the experimental analysis did contain extra errors. (In fact, some extra errors were discovered in different analysis that used Fault Classification and Automated Test Case generation by using Mutation Testing constant subject programs in other research that used the same subject programs. It is an important feature of the technique that the unknown errors do not hinder the technique; however, the model only captures the errors that are removed between the versions.

The mutation testing is a fault based testing strategy that measures the quality of testing by examining whether the test set, test input data used in testing can reveal certain type of faults. Mutation testing helps testers create test data by interacting with them to strengthen the quality of the test data. Faults are introduced into programs by creating many versions of the software, each containing one fault. Test cases are used to execute these faulty programs with the goal of causing each faulty program to produce incorrect output (fail). Hence the term mutation; faulty programs are mutants of the original, and a mutant is killed when it fails. When this happens, the mutant is considered dead and no longer needs to remain in the testing process because the faults represented by that mutant have been detected.

HEURISTIC CLASSIFICATION phenomenon as a member of a known class of objects, events, or processes. Typically, these classes are stereotypes that are hierarchically organized, and the process of identification is one of matching observations of an unknown entity against features of known classes. A paradigmatic example is identification of a plant or animal, using a guidebook of features, such as coloration, structure, and size. Heuristics of this type reduce search by skipping over intermediate relations (this is why we dont call abstraction relations *heuristics). These associations are usually uncertain because the intermediate relations may not hold in the specific case.

2. Related Work

Ferrari et al. known some fault varieties for facet headed programs that are extend during this paper. They additionally known a group of mutation operators associated with the faults. They outline the operators for aspectJ language and propose generalize mutation operators for different facet headed languages [5].Yves presents a tool named Aj Mutator for mutation analysis of purpose - cut descriptor. During this paper they implement purpose cut connected mutation operators that have been known within the previous analysis. This tool leverages the static analysis by the compiler to find the equivalent mutants mechanically [21]. Romain Delamare proposes a check driven approach for the event and validation of purpose cut descriptor. They designed a tool named recommendation Tracer that is employed to specify the expected be a part of points. To Validate the process; they additionally develop a tool that injects faults into purpose cut descriptors [50]. Alexander identifies key problems associated with the systematic testing of side bound programs. They develop a candidate fault model for side bound programs and derive testing criteria from the candidate fault model [12, 13, and 47]. Prasanth proposes a framework that mechanically finds the strength of purpose cuts and determines the various versions of the expression to settle on the right strength by the developer. Supported similarity live, the framework mechanically selects and rank completely different versions of purpose cut expression [10, 23]. Xie and Zhao developed a framework, named Aspectra, that mechanically generate the check inputs for testing grammatical relation behavior. Aspectra defines and measures grammatical relation branch coverage [49]. Wedyan and Ghosh gift a tool for activity be a part of purpose coverage from per recommendation and per category. This tool relies on AspectJ and Java computer memory unit code. This tool implements the framework that is given by Xie and Zhao [11]. Mortensen and T. Alexander use the static analysis of a side with during a system to settle on the white box criteria for a side like statement coverage, context coverage and Def-use coverage. They additionally give a collection of mutation operators associated with purpose cut and side precedence [12]. The approaches on top of don’t squarely address the primary issue that causes standard systems to be slow: interpretative execution. As noted antecedently, the overhead of compiling several mutant programs outweighs the advantage of increased fastness. Compiler-integrated [20] program mutation seeks to avoid excessive compilation overhead and yet retain the advantage of compiled speed execution. In this method, the program underneath check is compiled by a special compiler. Because the compilation method return, the results of mutations square measure noted and code patches that represent these mutations square measure ready. Execution of a specific mutant requires solely that the acceptable code patch be applied previous to execution. Mending is cheap and therefore the mutant executes at compiled-speeds. Sadly, crafting the required special compiler is a fashionable enterprise. Modifying Associate in nursing existing compiler reduces this burden somewhat; however the task remains technically exacting. Moreover, for every new laptop and operating system setting, this task should be recurrent.

In this system each node independently constructs its own local intrusion detection model according to its own data. There have been many survey of the field Dynamic DIDS. In particular Waiting Hu et al. [8] provide a comprehensive review of the online Adaboost-Based parameterized methods for Dynamic distributed network Intrusion detection which contain two models; Local Model and Global Model. Fig.2 gives an overview of framework that consists of the local models, and global models.

1) Local Models: Local model is constructed into each node by using weak classifiers and Adaboost-based training. So that each node contains a parametric model that consists of the parameters of the weak classifiers and the ensemble weights.
2) Global Models: It is constructed by combining all local parametric models by using PSO and SVM based algorithms. Global models are used to update local models and then updated models are shared by other nodes.
3) First we extract the features of incoming packets. There are total 41 features, all this features are grouped together and then passed to the SVM algorithm. After that PSO search the optimal results in KDD Dataset and output is created as a normal data or malicious data.

3. System Architecture

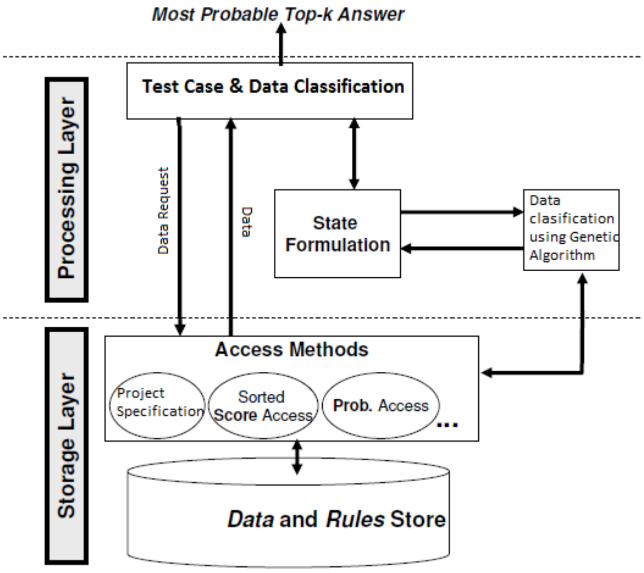

Figure 1: System Architecture

Software testing is very crucial part of software engineering. If the testing of software is not appropriate then there is no guarantee of the quality of software product. With the help of testing process we can ensure that software realize all the required functions as well as check the performance of software. The testing process must be done with the intention of finding bugs in the programs. In the software development life cycle, testing activities starts from requirement phase and goes along with all the intermediate process of development. Along with quality process, testing is the only activity which is carried out even after the development. Testing is necessary to develop any software system. Generally testing cost half of the total cost of software development and maintenance. There are two types of software testing first structural testing and second functional testing. Mutation testing or fault based testing is an example of structural testing for assessing the quality of software. In mutation testing we inject the fault in the original program and test same as the original program and compare the result of both programs to check the quality of program. These faulty programs are called mutants. For example, suppose a program is written to add two numbers i.e. c=a+b, for this program mutants are c=a-b, c=a*b, c=a/b. If output of both programs is not same on the same test cases i.e. c, then the mutant is killed. If all the mutants are killed, functionality of the program is good and test data is adequate. On the other hand, if the outputs of both programs are same that means mutants are alive. Mutants may be alive because of the test data is unable to distinguish the mutants or equivalent mutants. To distinguish the programs from its mutants, we need effective test cases to find faults in the program. Like other testing techniques, the efficiency of mutation testing depends on finding faults. Any mutation system can have these faults and this paper attempts to identify almost all the faults from such mutation system which is designed to represent these faults. These faults are implemented in the form of mutation operators. Effectiveness of mutation testing is depends on the quality of mutation operators. Mutation testing is not new but with respect to AOP it is new.

4. Framework of System

While working with the implementation of this paper, previous study shows how to produce a model of error correlated properties. This preprocessing step is run once, offline. The model is mechanically created from a collection this whole method is machine-controlled. The model is used as an input in Figure 1. Properties that appear only in non-faulty code are ignored by the machine learning step of programs with known errors and corrected versions of those programs. First, program analysis generates properties of programs with faults, and programs with those faults removed. Second, a machine learning algorithmic rule produces a model from these properties. Rectangles represent tools, and ovals represent tool inputs and outputs. This whole method is machine-controlled. The model is created by the technique of Figure.1 get that output in precisely faulty programs are tagged as fault-revealing, properties that seem in each faulty and non-faulty code are tagged as non fault-revealing, and properties that seem solely in non-faulty code aren’t used throughout training.(Figure.2)

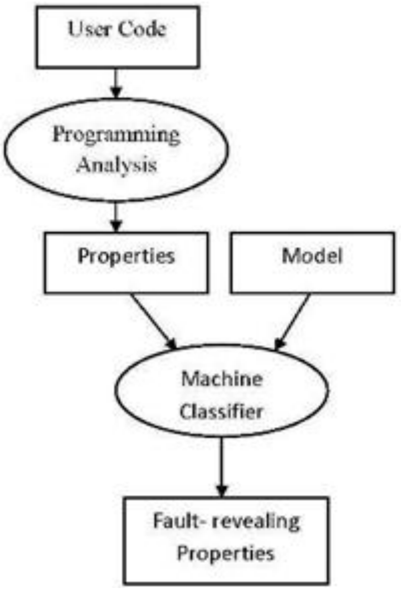

Figure 2: Finding likely fault revealing program properties using model

Detecting Faults Figure.3 shows how the Fault Invariant Classifier runs on code. First, a program analysis tool produces properties of the computer program.

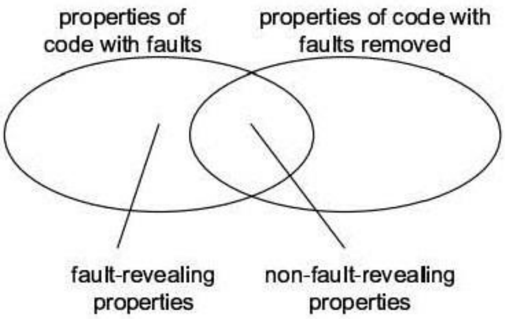
properties of code with faults
properties of code with faults removed
fault-revealing properties
non-fault-revealing properties

Figure -2: Fault-revealing program properties are those that appear in code with faults, but not in code without faults.

Second, a classifier ranks each property by its likelihood of being fault-revealing. A user who is curious about finding latent errors will begin by examining the properties classified as possibly to be fault-revealing.

Since machine learners are not guaranteed to produce perfect Models, this ranking is not guaranteed to be perfect, but examining the properties labeled as fault-revealing is more likely to lead the user to an error than examining randomly selected properties. The user only needs one fault revealing property to detect an error, so the user should examine the properties according to their rank, until an error is discovered, and rerun the tool after fixing the program code.

1. Log Generation
2. Filtering
3. Identification
4. Action

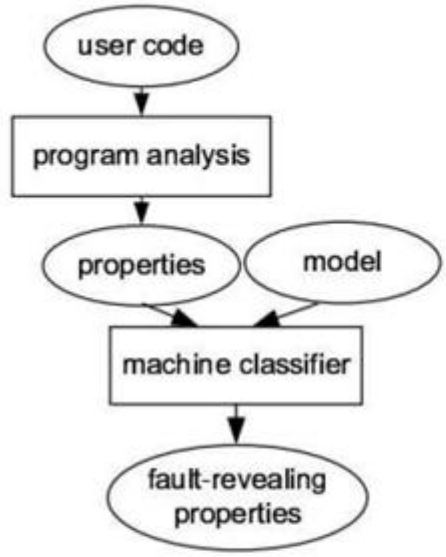

Figure 3: Finding likely fault-revealing program properties using a model.

MODULE 1: LOG GENERATION

These modules are given in to the following Figure 4. The first module going to work on the Log Generation of programs in which we are going to find out the Faults. Those programs also need to set program log properties according to which are found in the particular input program.

1. Setting Program Log Properties:

Logging facility has two parts: a configuration file, and an API for using logging services. Its a decent way, and is dead fine for straightforward and moderate work which we wants. (For complicated work wants, you may think about the various ways as an alternate.) Log entries are often sent to those Destinations, as either straightforward text or as XML:

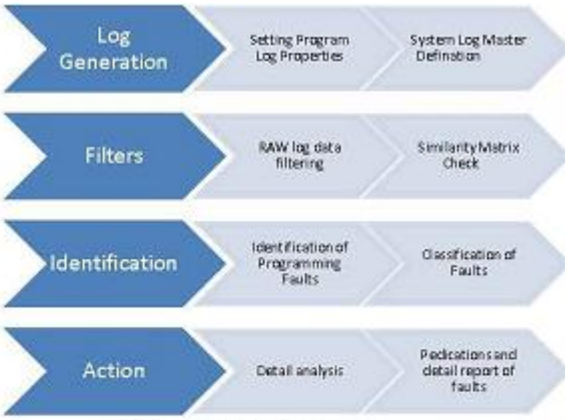
Log Generation
Setting Program Log Properties
System Log Master Definabion
Filters
RAW log data filtering
Similarity Matrix Check
Identification
Identification of Programming Faults
Classification of Faults
Action
Detail analysis
Pedications and detail report of faults

Figure 4: Modules

1. The Console
2. A File
3. A Stream
4. Memory
5. A TCP socket on a remote host

The Level category defines seven levels of work enlightenment: FINEST, FINER, FINE, CONFIG, INFO, WARNING, SEVERE ALL and OFF are outlined values similarly as Here is one type of victimization these Levels in code upon start up, use CONFIG to log configuration parameters throughout traditional operation, use information to log high-level ”heartbeat” data once bugs or essential conditions occur, use SEVERE Debugging data may default to FINE, with FINER and FINEST used often, in keeping with style. There’s flexibility in however work levels may be modified at run time, while not the requirement for a restart:

1) Simply change the configuration file and call Log Manager.
2) Or change the level in the body of your code, using the logging API; for example, one might automatically increase the logging level in response to unexpected events. Levels are attached to these items:
3) An originating logging request (from a single line of code) A Logger (usually attached to the package containing the Above line of code)
4) A Handler (attached to an application) One of the major challenges in pattern recognition problems is the feature extraction process which derives new features from existing features or directly from raw data in order to reduce the cost of computation during the classification process, while improving classifier efficiency. Most current feature extraction techniques transform the original pattern vector into a new vector with increased discrimination capability but
5) Lower dimensionality. This is conducted within a predefined feature space, and thus, has limited searching power.

5. Mathematical Model

Mathematical Model for Proposed Work

Assumptions:

S: System; A system is defined as a set such that:
S = {I, P, O}.

Where,
U: Set of users
I: Set of Input.
O: Set of output.
P: Set of Processes.

INPUT SET DETAILS:

1. *PHASE 1: DocumentOrCodeUpload.*
   Ir= {sourcecode: i1
   }

2. PHASE 2: CodeAnaysis
   IV= {sourcecode*: i1*
   }

Process Set Details

1. *PHASE 1: DocumentOrCodeUpload.*
P1= {DocumentOrCode_Read:P11,
DocumentOrCode_Save:P12}

2. PHASE 2: CodeAnaysis
P2= {code computation: p21,
Code processing:p22,
Mutant_analysis: p23}

3. PHASE 3: Result

P3= { SR_ Mutanats: p31,
SR_ Code_computation: p32,
SR_ Result: p33}

Output Set Details:

1. *PHASE 1: REGISTRATION.*
O1= {userid: o11,
Password: o12}

2. PHASE 2: QUERT PROCESSING
O2= {dataClassification: O21}

3. PHASE 3: Result
O3= {DR_Statistic: o31,
DR_Result: o32}

6. Experimental Results

ACCORDING TO C++ PROGRAM ANALSYIS, Four of the six mutation types applied produced mutants hich survived the test set: CONS, EDT, OAAN, and ORRN. This data suggest that SSDL was ineffective due to the software's achieving statement-level coverage. This data provides sufficient evidence to reduce the mutant subset further is questionable. As previously indicated, the work done on selective mutation identified a similar subset of mutant operators as being sufficient to achieve almost full.

Table 8.1: Results

Mutant Types	Total Mutants	Compiled Mutants
CONS	60	58
OAAN	85	83
ORRN	59	58
EDT	76	75

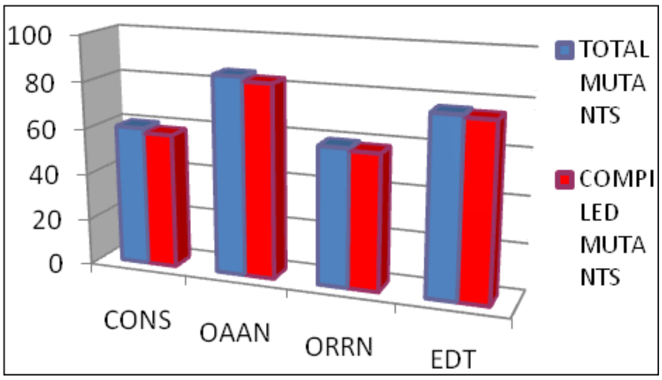

Figure 6: Mutant Types vs Total Mutants and Compiled Mutants

7. Conclusion

Thus proposed system is intended to design distinctive structures for the current programmed experiment era approaches, which presents a characterization of assessment methodologies. It's in view of sorts of utilizations, elements of programming we need to test, systems many-sided quality, and different components.

In this paper we represented different fault classification frameworks for the existing automatic test case generation approaches, and also have a brief look at each one. We described how to evaluate generated test cases, and introduce a classification of evaluation approaches. They based on types of applications, features of software we want to test, techniques complexity, and other features. Although there have been lots of researches on automatic test case generation problem, but for real world systems more researches are still needed.

References

[1] Weiming Hu, Jun Gao, Yanguo Wang, Ou Wu, and Stephen Maybank," Online Adaboost-Based Parameterized Methods for Dynamic Distributed Network Intrusion Detection," IEEE TRANSACTIONS ON CYBERNETICS, VOL. 44, NO. 1, JANUARY 2014

[2] Weiming Hu, W. Hu, and S. Maybank, "Adaboost-based algorithm for network intrusion detection," IEEE Trans. Syst., Man, Cybern., Part B: Cybern., vol. 38, no. 2, pp. 577–583, Apr. 2008.

[3] D. Denning, "An intrusion detection model," IEEE Trans. Softw. Eng., vol. SE-13, no. 2, pp. 222–232, Feb. 1987.

[4] Yan-guo Wang, Xi Li, and Weiming Hu: "Distributed Detection of Network Intrusions Based on a Parametric Model".

[5] S. Mabu, C. Chen, N. Lu, K. Shimada, and K. Hirasawa, "An intrusiondetection model based on fuzzy class-association-rule mining using genetic network programming," IEEE Trans. Syst., Man, Cybern., Part C: Appl. Rev., vol. 41, no. 1, pp. 130–139, Jan. 2011.

[6] J. Zhang, M. Zulkernine, and A. Haque, "Random-forests-based network intrusion detection systems," IEEE Trans. Syst., Man, Cybern., Part C:Appl. Rev., vol. 38, no. 5, pp. 649–659, Sep. 2008

[7] Yamille del Valle, Ganesh Kumar Venayagamoorthy, Salman Mohagheghi, Jean-Carlos Hernandez, "Particle Swarm Optimization: Basic Concepts, Variants and Applications in Power Systems" IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, VOL. 12, NO. 2, APRIL 2008

[8] J. Kennedy, "Particle swarm optimization," in Proc. IEEE Int. Conf. Neural Netw., 1995, pp. 1942–1948.

[9] Z. Zhang and H. Shen, "Online training of SVMs for real-time intrusion detection," in Proc. Adv. Inform. Netw. Appl., vol. 2, 2004, pp. 568–573.

[10] Muhammad Qasim Ali, Ehab Al-Shaer, and Taghrid Samak, "Firewall Policy Reconnaissance: Techniques and Analysis" IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, VOL. 9, NO. 2, FEBRUARY 2014

[11] Purvag Patel, Chet Langin, Feng Yu, and Shahram Rahimi, "Network Intrusion Detection Types and Computation" (IJCSIS) International Journal of Computer Science and Information Security, Vol. 10, No. 1, January 2012

[12] D. Smallwood and A. Vance, "Intrusion analysis with deep packet inspection: Increasing efficiency of packet based investigations," in Proc. Int. Conf. Cloud Service Computing, Dec. 2011, pp. 342–347.

[13] W. Lee, S. J. Stolfo, and K. Mork, "A data mining framework for building intrusion detection models," in

Proc. IEEE Symp. Security Privacy, May 1999, pp. 120–132.

[14] S. Mukkamala, G. Janoski, and A. Sung, “Intrusion detection using neural networks and support vector machines,” in Proc. Int. Joint Conf. Neural Netw., vol. 2. 2002, pp. 1702–1707.

[15] B. Pfahringer, “Winning the KDD99 classification cup: Bagged boosting,” SIGKDD Explorations, vol. 1, no. 2, pp. 65–66, 2000.

[16] W. Lee and S. J. Stolfo, “A framework for constructing features and models for intrusion detection systems,” ACM Trans. Inform. Syst. Security, vol. 3, no. 4, pp. 227–261, Nov. 2000.