Fault Model For Cause Effect Graph Testing

Anshul Mehta¹, Mr. Sachin Shrivastava², Gagan Thakral³

¹,²Dept. Computer Science, S.C.E.T. Palwal, India
³Computer Science, K.E.C.Ghaziabad, India

Abstract— Cause-effect graph is a directed Graph that maps a set of causes to a set of effects. The Causes may be thought of as input and the effect is thought of as the output. Cause-effect graphs can be used for specifying safety critical systems including avionics control software that are often intended to satisfy Boolean expressions. While Boolean expressions are useful to model predicates and complex conditions for state transitions, it is also true that they are very error prone to introduction of faults. Even though many Boolean specification based testing techniques have been proposed to detect faults of implementations of such specifications, there is almost no research about experimental investigation of the effectiveness of testing techniques with cause-effect graphs. In this paper, we present a new fault model which encompasses a variety of fault classes being hypothesized on the cause-effect graph. We have developed a tool to generate faults according to the fault model and to determine if a testing technique can detect those faults.

Keywords— Cause Effect Graph, Test Cases, Decision Table, Fault Classes.

I. INTRODUCTION

A key aspect of a fault model is to consider all possible faults that a programmer may commit. We say that a test set is adequate with respect to a fault model if it can detect all faults considered in the fault model. An ideal set of faults ensures that adequacy with respect to this set implies correctness. However, it is impossible to construct an ideal set of faults for any modeling (or programming) language because some types of faults made by human may be impossible to predict in advance. Mutation analysis addresses this problem by the so-called competent programmer hypothesis [10]. The hypothesis states that the program under test has been written by a competent programmer or designer. Therefore, if the programmer is not correct, it differs from a correct one by at most a few small faults. Based on this hypothesis, fault models try to designate a set of faults that ensures correctness with relatively high probability.

Various fault models have been developed and used to investigate testing effectiveness [5, 6, 11-14]. These studies on fault models are based on Boolean expressions. Some models require Boolean expression to be in DNF. The fault detection capability is measured using explicit fault classes which represent different types of error that can occur in a Boolean expression.

However, it is unlikely to specify the requirements with Boolean expressions directly. It is more likely that certain visual modeling languages such as the cause-effect graph are first used for modeling the requirements and then translated to Boolean expressions. In order to precisely model faults committed by human while modeling the requirements, it is necessary to deal with specification languages used for modeling the requirements directly. This observation motivated us to construct a fault model for the cause-effect graph rather than for Boolean expressions.

II. CAUSE-EFFECT GRAPH

A. Introduction

The Software testing technique Cause – Effect Graph was made-up by Bill Elmendorf of IBM in 1973 [1][2]. Instead of the test case designer trying to manually determine the right set of test cases, modeled the problem using a cause - effect graph, and the software that supports the
technique. Cause Effect Graph creates a relation between the outputs (conditions) and the inputs (actions).

A cause effect graph in software testing is an intended for graph that maps a number of conditions to a number of actions. The techniques in the past introduced regarding the different input data as self-determining, and the input value each one considered independently for producing test cases. The different inputs and their effects on the outputs are not clearly measured for test case design [1] [2] [5]. In precedent only some case studies were at hand wherever Cause effect graph application had taken consideration. Such are Withdrawal at ATM, Purchase order of Different equipment, Employee payroll system, some are with designed test cases and some are with test cases [3]. But in this thesis a new case study had chosen with outcome as test cases for the triangle problem as shown in further stages in this thesis.

A Cause-Effect Graph in Software Testing has been defined as: “It is a directed Graph that maps a set of causes to a set of effects. The Causes may be thought of as input and the effect is thought of as the output”. Usually the graph shows the nodes representing the causes on the left hand side and the nodes representing the effects on the right hand side. There may be intermediate nodes in between that combine inputs using logical operators such as AND, OR, Not etc.

It is a Software testing technique that helps us to select, in a systematic way, a high-yield of test-cases. It considers only the desired external behavior of a system [7]. “This is a testing technique that aids in selecting test cases that logically relate causes (inputs) to effects (Outputs) to produce-test-cases.

B. Steps Used In Caused Effect Graph

The Following Steps [4] are used to draw a Cause-Effect Graph.
Step1: For a module, identify the input conditions (Causes), and actions (output).
Step2: Develop a Cause-Effect Graph.
Step3: Transform Cause-Effect graph into a Decision Table.
Step4: Convert Decision table rules to Test cases. Each Column of the decision table represents a test case.
C. Basic Symbol Used In Cause-Effect Graphs

Furthermore, a cause-effect graph can specify constraints among causes. Figure 2 shows the constraints expressed in cause-effect graphs.

D. Derivation Of Test Cases

The following process is used to derive test cases:

i. The specification is divided into “Workable pieces”. For instance, when testing a timesharing system, a “Workable piece” might be specification for an individual command.

   ii. The causes and effects in the specification are identified.

   • Causes: Distinct conditions
   • Effects: An Output Condition or a system Transformation.

   iii. Assign a unique number to each cause and effect.

   iv. The semantic content of the specification is analyzed and transformed into a Boolean Graph linking the Causes and Effects. This is the Cause effect graph.

   v. The graph is annotated with constraints describing combinations of causes and/or effects that are impossible because of syntactic or environmental constraints.
vi. By methodically tracing state conditions in the graph, the graph is converted into a limited entry Decision –table. Each Column in the table represents a test case.

vii. The columns in the decision table are converted into test cases.

E. The cause-effect diagram can be used under these circumstances:
   i. To determine the current problem so that right decision can be taken very fast.
   ii. To narrate the connections of the system with the factors affecting a particular process or effect.
   iii. To recognize the probable root causes, the cause for an exact effect, problem, or outcome.

F. Benefits Of Making Cause-Effect Diagram
   i. It finds out the areas where data is collected for additional study.
   ii. It motivates team contribution and uses the team data of the process.
   iii. It uses synchronize and easy to read format to diagram cause-and-effect relationships.
   iv. Point out probable reasons of difference in a process.
   v. It enhances facts of the procedure by helping everyone to learn more about the factors at work and how they relate.
   vi. It assists us to decide the root reasons of a problem or quality using a structured approach.

III. A NEW FAULT MODEL FOR CAUSE-EFFECT GRAPH

A. Fault Classes
Various fault classes of Boolean expressions have been defined and studied. Usually the faulty is built from the original Boolean expression by one small syntactic change. Similarly, we define fault classes for the cause-effect graph. Suppose that a specification is given in the form of a cause-effect graph. We will use the cause-effect graph in Figure 3 to explain the fault classes proposed in this paper.

B. Fault model includes:
   i. Input Reference Fault (IRF): A cause (input) is replaced by 0, 1, or another cause which exists in the cause-effect graph. Figure 4 shows one IRF of the cause-effect graph in Figure 3 where C1 is replaced by C4. IRF is similar to the fault class, referred to as Variable Reference Fault (VRF) in the fault model for Boolean expressions [3, 7, 12]. IRF and VRF differ in a way of dealing with a nonsingular expression which contains multiple occurrences of a Boolean variable. For example, consider the nonsingular Boolean expression ab(b+c). Here, we have omitted the AND operator and used + to denote the OR operator. The corresponding cause-effect graph is shown in the left side of
Figure 5. A possible IRF of the cause-effect graph is shown in the right side of Figure 5 where b is replaced by a. However, there exists no VRF that corresponds to the IRF because VRF forces only one occurrence of a Boolean variable to be replaced by another at a time. A single fault in a cause-effect graph may correspond to more than one fault in its corresponding Boolean expression. However, it is unlikely to specify the requirements with Boolean expressions directly. It is more likely that certain visual modeling languages such as the cause-effect graph are first used for modeling the requirements and then translated to Boolean expressions. In order to precisely model faults committed by human while modeling the requirements, it is necessary to deal with specification languages used for modeling the requirements directly. This observation motivated us to construct a fault model for the cause-effect graph rather than for Boolean expressions.

ii. Input Negation Fault (INF): An occurrence of a cause is replaced by its negation. Figure 6 shows an INF of the cause-effect graph in Figure 3 where C4 is negated. INF is analogous to Variable Negation Fault (VRF) in the fault model for Boolean expressions. Note that INF deals with each occurrence of a cause rather than a cause itself. For example, the Boolean variable b in the left side of Figure 5 has two occurrences of it. These occurrences are represented by the edges from b to I1 and I2. In this case, only one occurrence of b is replaced by its negation when generating an INF.
iii. EXPRESSION NEGATION FAULT (ENF): An occurrence of an intermediate node is replaced by its negation. For example, the cause-effect graph in Figure 3 has I1, I2, I3, and I4 as its intermediate nodes. In this example, I1, I2, I4 have just one occurrence while I3 has two occurrences of it. As a consequence, five ENFs are generated. One of them is shown in Figure 7 where one occurrence of I3 is replaced by its negation.

iv. Operator Reference Fault (ORF): An occurrence of a logical operator AND is replaced by the OR operator or vice versa. Figure 8 shows an ORF generated by replacing the AND operator in I4 by the OR operator.
v. CONSTRAINT REFERENCE FAULT (CRF): This fault class concerns dependencies over causes. A CRF is generated by omitting a constraint or by replacing its constraint operator by another operator except the R constraint. For example, the E constraint in Figure 3 is omitted or is replaced by I or O constraint. Figure 9 shows a CRF generated by omitting the E constraint. In addition to these operator faults, CRF has operand faults. Each cause involved in a constraint is replaced by 0, 1, or another possible cause.

![Figure-9. A CRF Generated by Omitting the E Constraint](image)

Specifically, CRF has not been explicitly dealt with fault models for Boolean expressions even though input validation is considered very critical in areas including security testing. An application must be able to properly handle inputs coming from a variety of sources that are not built and arranged as expected. Of course, user inputs that are built and arranged as expected must be properly handled. To be considered safe and reliable, all applications must process valid inputs and respond to invalid inputs with reasonable tolerance. CRF can be used to assess testing techniques to investigate how well input validation is done. Problems due to incorrect input validation could lead to all sorts of problems and vulnerabilities such as buffer overflows or injection attacks.

IV. TOOL IMPLEMENTATION

We developed a tool for experimental investigation of testing software with cause-effect graphs. We will refer to the tool as “CEGTestingTool”. The goal of CEGTestingTool is to evaluate a cause-effect graph with various test sets generated according to certain testing strategies. Evaluation data are used to assess the effectiveness of a testing strategy for detecting specific faulty variations of the original cause-effect graph. CEGTestingTool runs on Java Runtime Environment 1.6 and has the textual interface. The tool starts by accepting two files. One input file is for the original cause-effect graph. For example, the input file format corresponding to the cause-effect graph in Figure 3 is shown in Figure 10.

All lines but the last two lines describe the structure of the cause-effect graph. The first column denotes either an effect or an intermediate node. One of the characters ‘|’, ‘&’, or ‘!’ may be placed in the second column. The character ‘|’ represents the “OR” operator, ‘&’ represents the “AND” operator, and ‘!’ represents the “NOT” operator. For example, the third line express that I3 is the result of logical “OR” of I2 and C3. Note that a node name can directly be placed in the second column without any such logical operators as seen in the first two lines.
The last two lines represent the constraints on the causes. The cause-effect graph has two constraints. One is the “Requires” constraint between C1 and C3. This is represented by “R C3 C1” where R denotes the “Requires” constraint which has C3 as its premise and C1 as its conclusion. The other is the “Exclusive-or” constraint among C2 and C4. This is simply represented by “E C2 C4” where E denotes the “Exclusive-or” constraint with its operands C2 and C4. Similarly, the operators “I” and “O” are used to denote the “Inclusive-or” and “One and only one” constraints, respectively.

The other input file is a test set used for evaluation. This file contains input values which are substituted into causes of a cause-effect graph under consideration.

After CEGTestingTool is supplied with a cause-effect graph file and a test set file, it generates all types of faults that it currently supports. CEGTestingTool proceeds to evaluate each test through each appropriate cause-effect graph and calculates various statistics based on the number of faulty cause-effect graphs that have results different from the original cause-effect graph. Such evaluation data are used to indicate the fault detection ability of a test suite for each fault class.

V. CONCLUSION

In this paper, we developed a fault model for the cause-effect graph and examined its applicability by evaluating Meyers’ approach and combinatorial testing. For the fault model proposed in this paper, we evaluated Meyers’ approach and combinatorial testing using pairwise, 3-way, and 4-way testing. The evaluation showed that Meyers’ approach was much superior to all types of combinatorial testing considered in this paper. We accept this as a natural consequence because Meyers’ approach is regarded as a white-box testing technique in that it exploits the internal structure of the cause-effect graph for test generation. On the contrary, combinatorial testing does not make use of any structural information of the cause-effect graph.

REFERENCES