Investigating Effectiveness of Software Testing with Cause-Effect Graphs

Insang Chung

Department of Computer Engineering,
Hansung University, Seoul, Republic of Korea
insang@hansung.ac.kr

Abstract

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. We show a case study where experimental assessment of testing effectiveness using two testing approaches, namely Meyers’ approach and combinatorial testing, has been carried to examine the applicability of our fault model based on the cause-effect graph.

Keywords: Cause-Effect Graph, Fault Model, Software Testing Effectiveness, Combinatorial Testing

1. Introduction

Many portions of the requirements of safety critical control software can be specified using Boolean expressions. As a result, many testing techniques like Modified Condition/Decision Coverage (MCDC), Branch Operator Strategy (BOR), and Basic Meaningful Impact Strategy (BMIS) have been proposed to generate tests from Boolean specifications [1-3]. This type of testing is usually referred to as predicate testing. Some predicate testing strategies can guarantee to detect certain type of faults.

Boolean expressions can be visually represented by cause-effect graphs. The cause effect graph was originally developed for hardware testing which was adapted for software specification and test generation [4]. It focuses on modelling logical relationships among program input conditions and output conditions. Even though the cause-effect graph is considered very promising in testing implementations that are intended to satisfy Boolean specifications, most of predicate testing strategies use directly Boolean expressions instead of cause-effect graphs in order to generate tests. As a consequence, cause-effect graphs have to be converted to Boolean expressions in particular formats such as disjunctive normal form (DNF) for existing predicate testing strategies to be applicable [5, 6]. A survey on most predicate testing is presented by [7].

One testing approach based on cause-effect graphs was proposed by Meyers [4]. Meyers’ approach constructs a decision table from a cause-effect graph and then
generates tests from the decision table. In order to avoid a large number of tests, Meyers applied some heuristics while deriving a decision table. Many predicate testing proposed previously belong to fault-based testing. In fault-based testing, certain types of faults that may be committed by programmers are firstly hypothesized, and then tests are developed so that those hypothesized faults can be detected [8]. Faults which fault-based testing methods presume are directly related to Boolean expressions. Thus, operators which are designed to model the potential faults need a direct representation of Boolean expressions to generate faulty expressions.

However, we observe that cause-effect graphs are more useful for modeling requirements than Boolean expressions. It is often the case that requirements are first modelled by cause-effect graphs and then Boolean expressions are produced from the cause-effect graphs for analysis and test. This observation has motivated our research to build a new fault model for the cause-effect graph rather than Boolean expressions. The new fault model tries to encompass all possible faults which may be committed by a person while modelling requirements with the cause-effect graph. We developed a tool to generate complete sets of faults in cause-effect graphs for several specific types of faults proposed in this paper. The tool can detect if a test set revealed a specific fault or not by comparing the results of the original cause-effect graph and faulty cause-effect graphs. We used the tool to conduct an experimental study to investigate effectiveness of Meyers’ approach and combinatorial testing [9].

The main contributions of this paper are summarized as follows:

- A new fault model for facilitating the development of fault-based testing methods based on cause-effect graphs and investigating effectiveness of various testing approaches.
- A tool that generates faulty cause-effect graphs for the proposed fault model. Without the tool, the investigation would have been tedious and error prone due to a large number of tests and faulty cause-effect graphs.
- A case study where Meyers’ approach and combinatorial testing are assessed to measure their fault detection capability. Each fault class represents a different type of errors that can occur in a cause-effect graph. This is similar to mutation operators applied to logic expressions in mutation testing [10].

The paper is organized as follows. Section 2 presents a brief overview of the cause-effect graph. Section 3 describes a new fault model for the cause-effect graph and introduces new fault classes which are specific for the cause-effect graph. Additionally, a tool for generating faulty cause-effect graphs for the proposed fault model and evaluating effectiveness of test suites is presented. Section 4 presents the results of conducting experimental investigations where Meyers’ approach and combinatorial testing have been evaluated to examine the applicability of our fault model based on the cause-effect graph. Finally, Section 5 summarizes and concludes the paper.

2. Cause-Effect Graphs

A cause-effect graph is originally developed for hardware testing, which is adapted to software testing. It specifies only the desired external behavior of a system by logically relating causes to effects to produce test cases. A cause represents a distinct input condition that brings about an internal change in the system. An effect represents an output condition, a system transformation or a state resulting from a combination of causes. Basic symbols used in cause-effect graphs are shown in Figure 1.
Each node has the value 0 or 1. The identity relation states that C is equivalent to E. That is, if C is 1, E is 1 or we can say if C is 0, E is 0. The NOT relation states that if C is 1, E is 0 and vice-versa. The OR relation states that if C1 or C2 is 1, E is 1 else E is 0. Similarly, the AND relation states that if both C1, and C2 are 1, E is 1; else E is 0. The AND and OR relations are allowed to have any number of inputs.

![Identity, NOT, AND, OR diagrams](image)

**Figure 1. Basic Elements of Cause-Effect Graphs**

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

![Exclusive-or, Inclusive-or, Requires, One and only one, Mask diagrams](image)

**Figure 2. Constraints of Cause-Effect Graphs**

There are various constraints among causes: E (Exclusive-or), O (One and only one), I (Inclusive-or), and R (Requires). The exclusive-or constraint states that at most one of the causes C1, C2, and C3 can be 1, i.e., they cannot be simultaneously. The Inclusive-or (at least one) constraint states that at least one of the causes C1, C2 or C3 must be 1. That is, all cannot be 0 simultaneously. The One and only one constraint states that only one of the causes C1 or C2 can be 1. The Requires constraint states that if cause C1 is 1, then cause C2 must be 1. The E, I, and O constraints can be related with any number of causes. In contrast to these constraints on causes, there is one constraint on effects known as Masking (M). The masking constraint states that if effect E1 is 1 then effect E2 is 0.
3. A New Fault Model for Cause-Effect Graph

3.1. Motivation

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.

3.2. 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.

![Figure 3. An Example Cause-Effect Graph](image-url)
Our current fault model includes:

- **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 modelling 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.

- **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.

![Figure 4. An IRF Generated by Replacing C1 by C4](image-url)
Figure 5. A Cause-Effect Graph corresponding to the Nonsingular Boolean Expression \(ab(b+c)\) and an IRF Generated by Replacing \(b\) by \(a\)

Figure 6. An INF Generated by Replacing One Occurrence of \(C4\) by Its Negation

Figure 7. An ENF Generated by Replacing One Occurrence of \(I3\) by Its Negation

- **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.
• **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.

![Figure 8](image)

**Figure 8. An ORF Generated by Replacing the AND Operator in I4 by the OR Operator**

• **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](image)

**Figure 9. A CRF Generated by Omitting the E Constraint**

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.

### 3.3. Tool Implementation

We developed a tool for experimental investigation of testing software with cause-effect graphs. We will refer to the tool as “CEGTester”. The goal of CEGTester 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. CEGTester 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.

![Figure 10. An Example of an Input File](image)

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 CEGTester is supplied with a cause-effect graph file and a test set file, it generates all types of faults that it currently supports. CEGTester 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.
4. A Case Study

Using CEGTester, we investigated effectiveness of two testing approaches: Meyers’ approach and combinatorial testing. Combinatorial testing is different from Meyers’ approach in that test generation can be carried out independently of the structure of the cause-effect graph. On the other hand, Meyers’ approach considers the structure of a cause-effect graph to produce tests.

Meyers’ approach starts with identifying causes and effects in the requirements. In the next step, a cause-effect graph is constructed by considering the relationships extracted from the specification. The graph can be annotated with constraints describing combinations of causes and effects that are impossible because of syntactic or environmental constraints. Finally, the graph is converted into a decision table. Each column in the decision table represents a combination of input values, and hence a test. During the process of converting a cause-effect graph into a decision table, certain heuristics can be applied to avoid combinatorial explosion of tests.

The key insight underlying combinatorial testing is that many faults are caused by interactions between a relatively small numbers of input parameter values [9]. The most basic combinatorial testing, known as pairwise testing, generates tests which capture all possible combinations of pairs of input parameter values. For example, consider a system of 30 input parameters where each parameter can be assigned one of 10 values. Exhaustive testing would require the execution of $10^{30}$ input combinations. On the other hand, there are a total of 100 pairwise tests which capture all possible pairs of input values. Pairwise testing is based on the premise that most software faults can be captured by either single-value inputs or by an interaction between pairs of input values. Studies have shown pairwise testing to be a very practical and effective software testing criterion even though the size of test sets is dramatically reduced. Similarly, t-way (t=3, 4 …) combinatorial testing can be defined. For our investigation, we employed pairwise, 3-way, and 4-way combinatorial testing.

Table 1. Number of Generated Faults

<table>
<thead>
<tr>
<th>Number of generated faults</th>
<th>IRF</th>
<th>INF</th>
<th>ENF</th>
<th>ORF</th>
<th>CRF</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>340</td>
<td>25</td>
<td>18</td>
<td>14</td>
<td>177</td>
<td></td>
<td>574</td>
</tr>
</tbody>
</table>

We use the cause-effect graph of the DISPLAY command in [4] to evaluate the two testing approaches for the proposed fault model. The graph consists of 35 nodes. Among the nodes there are 18 causes and 7 effects. It has 17 “Requires” constraints and one “Exclusive-or” constraint.

We used CEGTester to generate faults. Table 1 shows the number of generated faults according to the five types of faults mentioned in Section 3.2. For the graph, we constructed test suites by using Meyers’ approach and combinatorial testing. We used the test suite in [4] for Meyers’ approach. The test suite has 38 input combinations. For combinatorial testing, we generated pairwise, 3-way, and 4-way combinatorial tests by using PICT [15]. PICT is a command line tool for combinatorial testing developed at Microsoft. PICT generated 10 pairwise tests, 25 3-way tests, and 65 4-way tests. These test suites were evaluated with mutation score used in mutation analysis. Mutation score is defined as “Number of faults detected/Number of all faults generated”.

Table 1. Number of Generated Faults
Figure 11 shows the total mutation scores and the mutation scores by fault classes for Meyers’ approach and pairwise, 3-way, and 4-way combinatorial testing. The total mutation score of Meyers’ approach is superior to all types of combinatorial testing. Moreover, Meyers’ approach is much superior to pairwise testing for all fault classes. The mutation scores of 3-way and 4-way testing for ENF and ORF are 100 which are the same as those of Meyers approach. The 4-way testing is also as effective as Meyers’ approach for INF.

We can see that CRFs are most difficult to be detected for all types of combinatorial testing. The average mutation score of pairwise, 3-way, and 4-way testing for CRF was (45.4+49.5+77.6)/3=57.5 while the mutation score of Meyers’ approach for CRF was 90.3. Surprisingly, the test suite generated by the 4-way was about 1.7 times larger than the one generated by Meyers’ approach. This result indicates that further research should be focused on how to improve the ability of detecting CRFs to order to use combinatorial testing effectively. This study can be valuable when specification is not available or its information is limited to testing.

Combinatorial testing generates tests irrespective of internal structure of specifications or programs. In the present case, specifications are given in the form of cause-effect graphs. Meyers’ approach exploits the structure of a given cause-effect graph to generate tests. On the other hand, combinatorial testing does not make use of any information in specifications for test generation. We think that this difference led to much better evaluation results for Meyers’ approach than for combinatorial testing.

We conducted another experiment to study how to improve the effectiveness of combinatorial testing for detecting CRFs. Many tests generated by combinatorial testing did not satisfy constraints on both original and faulty cause-effect graphs, leading to no discrimination between them. This investigation indicates that constraints on inputs play very critical role on detecting CRFs. For the reason, we augmented combinatorial testing with constraints on inputs. This augmentation was easily done because PICT allows us to specify constraints by using various constructs.

When augmenting with constraints among inputs, PICT generated 19 pairwise tests, 49 3-way tests, and 109 4-way tests. Removing tests which do not satisfy constraints may also eliminate certain interactions among inputs that need to be maintained. Thus, new tests need to be added to keep those interactions in place. Usually, more tests are added than tests removed. We can see that the test suites are bigger than those generated without considering constraints.
As shown in Figure 12, the mutation scores of all types of combinatorial testing considered in this paper were comparative to the mutation score of Meyers’ approach. More specifically, the mutation scores of pairwise, 3-way, and 4-way testing were 95, 96.6 and 97, respectively. This result is somewhat surprising because 3-way and 4-way testing perform better than Meyers’ approach which obtained 96.5 as its mutation score. Moreover, the difference of the mutation scores between pairwise testing and combinatorial testing is considered negligible even though the size of the test suite obtained by pairwise testing is half of the test suite obtained by Meyers’ approach. We can see a possibility of using pairwise testing as an alternative if the internal structure of a specification is not available, but constraints on inputs can be given.

![Figure 12. Mutation Scores Obtained by Considering Constraints](image)

5. Concluding Remarks

In this paper, we developed a fault model for the cause-effect graph and examined its applicability by evaluating Meyers’ approach and combinatorial testing. We also developed CEGTester, a new tool for experimental investigation of testing effectiveness with cause-effect graphs in order to avoid tedious and error prone tasks. The tool automatically generates all possible faulty cause-effect graphs for a given cause-effect graph for the five types of faults suggested in this paper. It uses mutation analysis for the assessment of effectiveness and produces mutation scores of test suites obtained by various testing criteria.

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.

A detailed investigation of the evaluation results showed that it is difficult to detect CRFs by combinatorial testing. Based on the results, we conducted another investigation where combinatorial testing was augmented to consider constraints among causes. In the investigation, pairwise testing was shown to be comparative to Meyers’ approach even though the size of the test suite of pairwise testing was half of that produced by Meyers’ approach. This indicates that if combinatorial testing is given information on constraints among inputs, it is possible to generate very effective tests.
without exploiting the structural information of specifications. We plan to develop a more effective testing method by taking this finding into consideration after more case studies for evaluating the proposed fault model are done.

Acknowledgements

This research was financially supported by Hansung University.

References

Author

Insang Chung, he is a Professor in Department of Computer Engineering, Hansung University, Seoul, S. Korea. He received his Bachelor of Engineering degree at Seoul National University, S. Korea in 1987. He also received MS and PhD in Computer Science from KAIST(Korea Advanced Institute of Science and Technology), S. Korea in 1989 and 1993, respectively. His research interests are in automated test data generation, formal techniques including model checking and testing process for safety-critical software such as automotive software. He has authored many refereed journals and conference papers about software testing.