Proceedings of the 6th Győr Symposium and 3rd Hungarian-Polish
and 1st Hungarian-Romanian Joint Conference on Computational Intelligence

Generation of Software Tests on the Basis of Cause-Effect Graphs

József Sziray

**Széchenyi University, Department of Informatics,
Egyetem tér 1, H-9026 Győr,
Hungary E-mail: sziray@sze.hu**

Abstract: Cause-effect graphs are applied for preparing efficient functional tests for software. A graph established on the basis of the software specification is required to be evaluated. The evaluation results in the test cases consisting of the Boolean-logic combinations of causes. A cause-effect graph is equivalent with a combinational logic network. This paper presents an exact algorithm for producing the test cases of the software. The algorithm applies a three-valued Boolean algebra, and is based on the successive justification of logic values in a combinational logic network, where the primary inputs are the causes, and the primary outputs are the effects. The computations are performed by traversing a decision tree, where backtracking is required if a decision leads to a logic contradiction. The main advantage of the algorithm is that it reduces the number of decisions to a great extent by using don't care values in the process. The calculation principle is comparatively simple. It is based only on successive line-value justification, and it yields an opportunity to be realized by an efficient computer program. The logic model introduced in the paper is completely general, in that it is applicable to any kind of cause-effect graphs, without any constraint. The final part of the paper is concerned with the computational complexity of the presented algorithm.

Keywords: Software testing, cause-effect graphs, three-valued Boolean algebra, combinational logic networks, line-value justification, computational complexity.

1. Introduction

Testing is an integral part of the software life cycle, including development and maintenance. In fact, this activity may easily take 50 % or more of the overall development cost [1]. Software testing is carried out at different levels throughout the entire software life-cycle. Testing starts with individual software components. Each component should be checked functionally and structurally. Testing is also necessary during the integration of software components to ensure that each combination of components is satisfactory. System and acceptance testing follow component and

integration testing [2]-[9]. The IEEE standard on software verification and validation (IEEE Std. 1059-1993) identifies four levels of testing. These are: component tests, integration tests, system tests, and acceptance tests [5].

In the case of functional testing, we are interested to examine what the software is supposed to do. The process is worked out from an input data perspective, where we subsequently see if the outputs (actions) of the software match the expected response values. On the other hand, structured testing concentrates on the internal structure of the source code, where the execution of the program statements, through branches, cycles and paths are examined.

The two fundamental approaches, i.e., functional and structural, can principally be applied at any of the levels of testing. However, there exists a practical constraint for the structural approach, which is the complexity of the code.

There are several methods available for designing test cases for a given software, both for the functional, and for the structural approach [2]-[9]. In this paper we are going to deal with a functional-oriented method, namely, the use of cause-effect graphs, which are also called Boolean graphs [2]-[3].

The concept of establishing and using such graphs was originally worked out by Elmendorf [2] and extended by Myers [3]. Here the concept is to convert the functional requirements to a formal specification of the same software. The process examines the semantics of the requirements and restates them as logical relationships between inputs and outputs. The inputs are called causes, and the outputs are effects. This process can be done in an intuitive way, where first we have to find the causes and then the corresponding effects. The next task is to establish the logic connections between the existence of the causes and the effects. Once we have the graph, it can be used to enumerate test cases for functional testing. The resulted test cases are not redundant; that is, one test case does not test functions that have already been tested by another case. In addition, the process finds incomplete and ambiguous aspects of requirements in the original software specification, if any exist. If so, the specification needs to be amended.

The obtained cause-effect graph is a Boolean graph reflecting the input-output relationships by way of Boolean logic. This type of representation can be considered as a formal description of the software behavior.

This paper presents an exact algorithm which can be applied for evaluating a given cause-effect graph. Here evaluation means the calculation of the Boolean values of the causes, as a result of the individual effects, selected each effect one by one. The obtained logic values of the causes will be used to form the test cases for functional testing. The proposed algorithm is exact in the sense that it results always in a solution if one exists.

2. The logic model of a cause-effect graph

It is known that each node of a cause-effect graph has a logic value derived from the Boolean algebra. That is why such a graph is also called Boolean graph. The logic value

of a cause is one (1), if and only if the cause is existent (i.e., it is present), otherwise it is zero (0), i.e., the cause is nonexistent (it is absent). The same applies to each effect, where an effect is the response of the software to the various causes.

The internal nodes of a Boolean graph are also associated with logic values, where, in addition a node can perform the following logic operations: AND, OR, NOT, and YES, where NOT means logic inversion, and the YES operation does not alter the actual value.

The AND operation of a node is denoted by a ^ sign, OR is denoted by a v, while logic inversion is represented by a ‖ sign, which is placed on the edge of the graph, this way performing inversion between the two connected nodes. The elements of a graph are depicted in Figure 1.

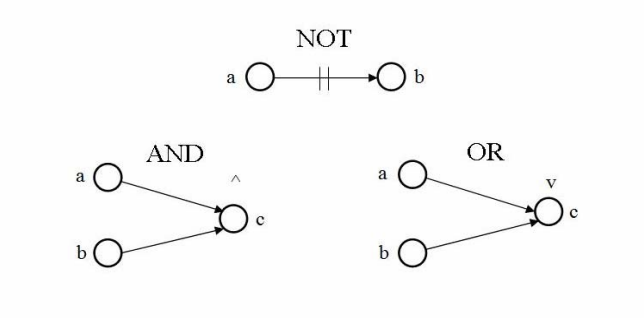

Figure 1. Logic operations in a Boolean graph

If we assign logic gates to these nodes, in accordance with their operations, then the Boolean graph will be transformed to an equivalent combinational logic network. In this network the primary inputs are the causes, while the primary outputs are the effects, while the internal nodes with logic operations will correspond to logic gates with the same operations. The direction of the graph edges are in accordance with the direction of the signal propagation.

In order to obtain the software tests each effect is associated with logic one, independent from the other effects, and the corresponding causes are to be determined, which result in the selected effect. The selection of the individual effects are done one-by-one, separately, where each effect is to be justified by the correspondent causes. The next section will show an algorithm for computing the logic values of the causes belonging to a given effect.

The calculations are performed within a three-valued logic system, where 0, 1 and don't care (d) values are involved. Here, don't care means that the actual logic value is indifferent: it can either be 0 or 1. The logic operations in this system are enlisted in Table 1, as follows:

Table 1. Logic operations

0 AND 0 = 0, 0 AND 1 = 0, 0 AND d = 0.
1 AND 0 = 0, 1 AND 1 = 1, 1 AND d = d.
d AND 0 = 0, d AND 1 = d, d AND d = d.

0 OR 0 = 0, 0 OR 1 = 1, 0 OR d = d.
1 OR 0 = 1, 1 OR 1 = 1, 1 OR d = 1.
d OR 0 = d, d OR 1 = 1, d OR d = d.

NOT 0 = 1, NOT 1 = 0, NOT d = d.

It can be seen that the above summary of operations defines a three-valued Boolean system.

3. The process of line-value justification

Logic values in a network are associated with the primary input and output lines, as well as with the output lines of the logic elements. The proposed calculation process is based on the so-called line-value justification, which was originally used in test generation of logic networks [10]-[12].

Suppose the lines of the network are numbered in an increasing order, starting from 1, and let the logic value of line i be denoted by v(i). In this notation, for a primary input and output it is also allowed to use v(i).

Line-value justification (or shortly line justification) is a systematic procedure with the aim of successively assigning input values to the logic elements, in such a way that they are consistent with each previously assigned value. The output value v(i) of a gate is said to be justified if the input values unambiguously result in v(i). It should be noted that for a given output value at a gate not only one input combination can be assigned, since there may be more than one possible choices. When performing this process the following viewpoints have to be taken into consideration:

- Only the determined logic values, 0 and 1, have to be traced back, i.e., these values are to be justified at the gate inputs. The value of d needs no justification, so it is unnecessary to trace it back.
- Since d does not require justification, it is worth assigning the minimum number of determined values to the gate inputs, while leaving the others at the value of d.

If two-input gates are considered, the possible choices are summarized below:

- AND gate, with an output value 1: Both inputs must be 1.
- AND gate, with an output value 0: One input is 0, the other is d.
- OR gate, with an output value 0: Both inputs must be 0.
- OR gate, with an output value 1: One input is 1, the other is d.
- If the number of inputs at a gate were more than two, it would increase the number of choices, but would not cause any difference in principle. The overall goal here is to assign as many don't cares as possible.

The line-justification procedure continues until all the necessary primary inputs have been reached. This is a decision process where contradictions may occur in the logic values at the network lines. In order to eliminate a contradiction we have to systematically change the previously made decisions. Here we have to return to the last decision which can be changed, then delete all the consequences of the last decision,

and continue by making a new decision. These steps are called backtracing or backtracking.

An example is shown in Figure 2, where the primary output value v(10) = 1 is to be justified. The assigned values in a decreasing line order are: v(8) = 0, v(9) = d, v(6) = 0, v(5) = 0, v(2) = 0, v(3) = 0. Here, v(5) requires v(1) = 1 and v(2) = 1 which is a contradiction.

Now we have to trace back the decision chain. The last modifiable decision is at v(10). The new decision is v(8) = d and v(9) = 0, which results in v(6) = 1, v(7) = 1, v(2) = 1, v(3) = d, and finally v(4) = 0. Since v(1) was not involved, its value is d. So the primary input vector justifying v(10) = 1 is

$$\overline{v} = (d, 1, d, 0),$$

which is shown in Figure 3.

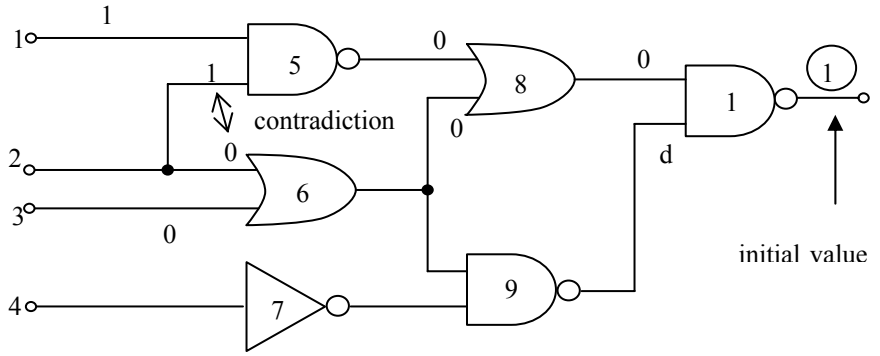

Figure 2. An example for line-value justification

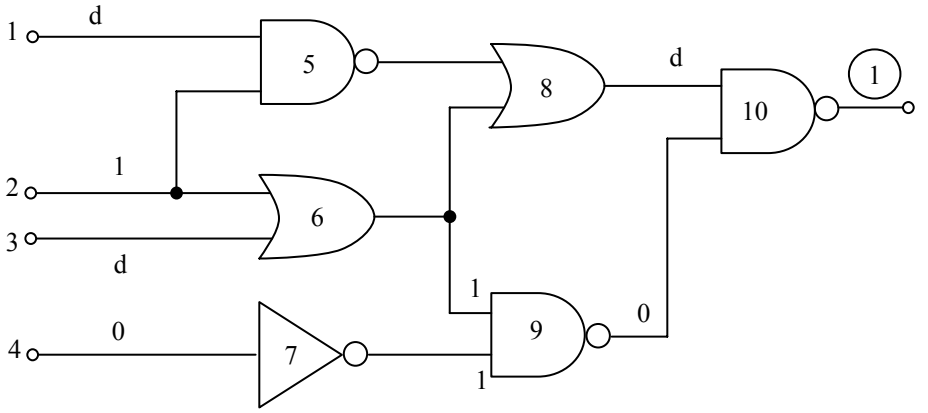

Figure 3. Assigned line values after back trace

4. Computational examples

In this section, two computational examples will be presented to illustrate the use of line-value justification. It is supposed that the cause-effect graphs have already been derived, thus they are considered to be available.

4.1. The first example

The first graph is shown in Figure 4. The equivalent logic network of the graph can be seen in Figure 5.

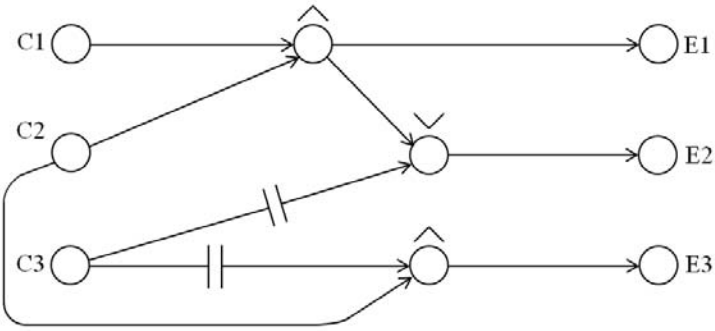

Figure 4. First example: A cause-effect graph

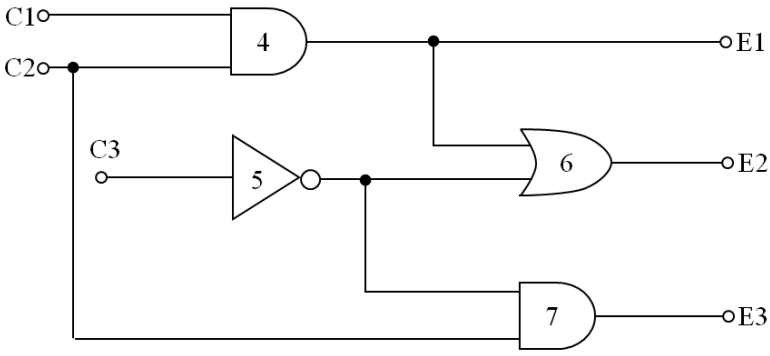

Figure 5. First example: The equivalent logic network

In general, there may be some constrains related to the occurrence of certain input conditions to a software. In our case, suppose that the following constraints are imposed:

One or two causes are always present, but the simultaneous presence of all the three causes is prohibited. It means that neither of the test inputs

(C1, C2, C3) = (0, 0, 0) and

$$(C1, C2, C3) = (1, 1, 1)$$

are allowed.

The justification process is carried out in the following way:

For E1 = 1:

$$C1 = 1, C2 = 1,$$

$$\text{and } C3 = d.$$

However, because of the constraint among the causes, the only consistent test vector is

$$(C1, C2, C3) = (1, 1, 0).$$

For E2 = 1:

$$v(4) = 1, \ v(5) = d,$$

$$C1 = 1, C2 = 1,$$

$$\text{and } C3 = d.$$

This result will yield again the test

$$(C1, C2, C3) = (1, 1, 0).$$

However, we have an other choice for v(6), i.e.,

$$v(4) = d, \ v(5) = 1,$$

$$C1 = d, C2 = d,$$

$$\text{and } C3 = 0.$$

Now the two other possible test vectors belonging to E2 are

$$(C1, C2, C3) = (0, 1, 0), \text{ and}$$

$$(C1, C2, C3) = (1, 0, 0).$$

For E3 = 1:

$$v(5) = 1, \ C2 = 1,$$

$$C3 = 0, \text{ and } C1 = d.$$

Here, C1 can have both 0 and 1, so the test vectors belonging to E3 are

$$(C1, C2, C3) = (0, 1, 0), \text{ and}$$

$$(C1, C2, C3) = (1, 1, 0).$$

After all, we have received three different input vectors which yield the complete set of cause-effect tests for the graph in Figure 3. These are

$$\overline{x}_1 = (1, 1, 0),$$

$$\overline{x}_2 = (0, 1, 0),$$

$$\overline{x}_3 = (1, 0, 0).$$

It can be seen that 1 belongs to all the three effects, 2 belongs to E2 and E3, while 3 belongs to E2 only.

4.2. The second example

The second network can be seen in Figure 6. This is the same example as Glenford Myers used in his fundamental book [3] (see page 58). (The logic network of the graph is also included in the book.) Here, Cause 1 (C1) means that the first input character is "A", while Cause 2 (C2) means that the first input character is "B". Since an input character at the same position cannot be an "A" and a "B" at the same time, it yields the constraint of prohibiting C1 = 1 and C2 = 1 to occur simultaneously.

The justification process is as follows:

For E1 = 1:

$$v(4) = 0,$$

$$C1 = 0, C2 = 0.$$

The correspondent test is

$$(C1, C2, C3) = (0, 0, d),$$

which yields

$$\overline{x}_1 = (0, 0, 0), \text{ and}$$

$$\overline{x}_2 = (0, 0, 1).$$

For E2 = 1:

$$v(4) = 1, \text{ and } C3 = 1,$$

$$C1 = 1, \text{ and } C2 = d.$$

Due to the constraint on C1 and C2, the assignment of C2 = 1 is prohibited, so the don't care value at C2 must be set to 0. The appropriate test is:

$$\overline{x}_3 = (1, 0, 1).$$

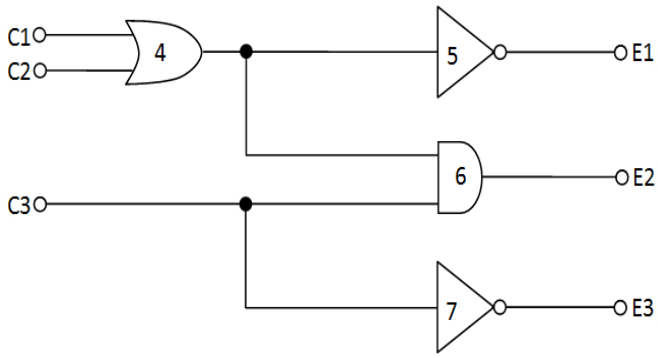

Figure 6. Second example: A cause-effect graph as a logic network from Myers

Since we have another choice at gate 4, i.e.,

$$C1 = d, \text{ and } C2 = 1,$$

an other valid test can be obtained:

$$\overline{X}_4 = (0, 1, 1).$$

Finally, for E3 = 1: $C3 = 0, C1 = d, C2 = d.$

For this case, the two possible test vectors are

$$\overline{X}_5 = (1, 0, 0), \text{ and}$$

$$\overline{X}_6 = (0, 1, 0).$$

5. Conclusions

The paper presented an algorithm for producing the logic conditions that result in the effects of a cause-effect graph belonging to a particular software. These conditions yield the test cases of the software. The algorithm applies a three-valued Boolean algebra, and is based on the successive justification of logic values in a combinational network, where the primary inputs are the causes, and the primary outputs are the effects. The computations are performed by traversing a decision tree, where backtracking is required if a decision leads to a logic contradiction.

As we have seen in Section 3, line-value justification is a systematic procedure with the aim of successively assigning input values to the logic elements, in such a way that they are consistent with each previously assigned value. Whenever a contradiction occurs in assigning a value, we backtrack to the last assignment that can still be altered. Here we make another choice and proceed with the computations. The justification process is carried out by traversing downward and upward on the corresponding decision tree. In worst case, the complete tree is to be traversed.

The main advantage of the algorithm is that it reduces the number of decisions to a great extent by using don't care values in the process. It means that the decision tree

with only true-false values is pruned to a great extent in our case, thus having significantly less branch in it.

The logic model introduced in the paper is completely general, in that it is applicable to any kind of cause-effect graphs, without any constraint.

The calculations using this three-valued logic can advantageously be organized and carried out on a computer, due to the following reasons:

- The storage requirement of the three logic values at the network lines is negligible.
- Computations among logic values are ab ovo fast and efficient.
- The data-base structure of a logic network is comparatively simple. Only the gate types and the input-output connections of the gates are to be encoded and stored. It should also be mentioned that the computations and backward tracing are carried out directly on this same network structure.

The amount of computations depends on the number of gates in the network. On the basis of a concrete network a decision tree is established, where the number of computational steps is greatly influenced by the number of necessary backtrackings within the tree. As known, the given computational problem belongs to the class of NP-complete problems (NP: nondeterministic polynomial) [12]-[13].

The exact complexity of the algorithms solving such problems cannot be determined in general, only an upper limit can be given, which is an integer number depending exponentially on the size of the actual problem. In our case, this size is the number of gates in the network.

References

[1] N. Storey: “Safety-Critical Computer Systems”, Addison-Wesley-Longman, Inc., New York, 1996.
[2] W. R. Elmendorf: “Cause-Effect Graphs in Functional Testing”, Technical Report TR-00.2487. Poughkeepsie, NY: IBM Systems Development Division, 1973.
[3] G. J. Myers: “The Art of Software Testing”, John Wiley & Sons, Inc., New York, 1979.
[4] B. Beizer: “Black-Box Testing, Techniques for Functional Testing of Software and Systems”, John Wiley & Sons, Inc., New York, 1995.
[5] J. F. Peters, W. Pedrycz: “Software Engineering, An Engineering Approach”, John Wiley & Sons, Inc., New York, 2000.
[6] Sh. L. Pfleeger: “Software Engineering, Theory and Practice”, Second Edition, Prentice-Hall, Inc., USA, 2001.
[7] P. C. Jorgensen: “Software Testing, A Craftsman's Approach”, Second Edition, CRC Press LLC, USA, 2002.
[8] C. Ghezzi, M. Jazayeri, D. Mandrioli: “Fundamentals of Software Engineering”, Second Edition, Prentice Hall, Pearson Education, Inc., USA, 2003.
[9] I. Sommerville: “Software Engineering, Eighth Edition”, Addison-Wesley, Pearson Education Limited, Harlow, England, 2007.

[10] M. Abramovici, M. A. Breuer, A. D. Friedman: “Digital Systems Testing and Testable Design”, Computer Science Press, USA, 1990.
[11] J. Sziray: “Test Calculation for Logic and Delay Faults in Digital Circuits”, IEEE Microprocessor Test and Verification Workshop, (MTV-06), Proceedings, pp. 20-29, Austin, Texas, USA, December, 2006.
[12] J. Sziray: “Test Design of Digital Systems”, Széchenyi University Press, Győr, Hungary, 2012.
[13] Harry R. Lewis, Christos H. Papadimitriou, “Elements of the Theory of Computation”, Prentice-Hall, Inc., USA, 1998.