European Journal of Computer Science and Information Technology

Vol.5, No.4, pp.11-24, August 2017

Published by European Centre for Research Training and Development UK (www.eajournals.org)

A CAUSE-EFFECT GRAPH SOFTWARE TESTING TOOL

Berk Bekiroglu

Department of Computer Science, Illinois Institute of Technology

ABSTRACT: *This paper explains analysis and implementation of a cause-effect graph software testing tool. The cause-effect graph software testing method and its application are described. The method of generating test cases from software specification is discussed. In addition, a new cause-effect graph testing tool is developed, and processes in the cause-effect graph software testing is explained with an example. Moreover, the coverage analysis of effect nodes is described.*

KEYWORDS: Software Testing Tools, Cause-Effect, Graph Software, Testing

INTRODUCTION

Software testing is an essential part of software development processes. It is the most expensive tasks during a software development. Software testing usually involves series of planned processes, and the aim of these processes is to find bugs in a system. All software systems have plenty of bugs, and the software testing decreases the number of these bugs. Although software testing methods can find critical bugs in software, there may be still significant numbers of bugs in well-tested systems. Thus, a bug-free system is impossible in practice. However, software bugs cannot be tolerable in some systems, such as safety-critical, and mission critical systems where the failure of systems can cause catastrophic results. For this reason, software testing becomes more important because of the consequences of failures for these types of systems.

Software testing methods are divided into two main categories, namely black box software testing and white box software testing. Black box software testing methods are based on software specifications. In addition, the program is considered as a closed box and is assumed the internal structure of the program is not known. Some important black box software testing methods are equivalence portioning, boundary value analysis, state transition and use case testing (Burnstein, 2002; Mayers, 2004; Nook, 2008; Lewis, 2009; Everett, 2007). Furthermore, the cause-effect graph software testing method is in this category because it is based on software specifications. On the other hand, white box software testing methods are performed with the knowledge of the internal structure of programs. Source code coverage methods such as statement coverage, decision coverage, and condition coverage are some of the most used white box software testing methods (Burnstein, 2002; Mayers, 2004; Nook, 2008; Lewis, 2009; Everett, 2007). During a software testing process, more than one software testing method can be used complementarily, and each additional software testing method can find new bugs in a system.

A cause-effect graph software testing considers combinations of input conditions when generating test cases. The software tester should determine input conditions, outputs, and constraints from software specifications. The cause-effect graph software testing may find some types of bugs that other black box software testing method cannot find.

Because of size and complexity of software systems, performing manual software testing is tedious, and some automation may be necessary (Burnstein, 2002). For this reason, software testing tools are developed to carry out these software testing methods. By automated software testing tools, software testing can be done more systematically and correctly. The cause-effect graph testing tool is one of these software testing tools which use the cause-effect graph software testing method. Cause-effect graph software testing tools generate decision tables from logical circuit designs. Then, they produce test cases from decision tables (Burnstein, 2002; Bender, 2008). Although many processes can be automated with software testing tools, cause-effect software testing tools still require many manual processes such as specifying input conditions and developing logical circuits which also require analyzing constraints on input conditions. So, the cause-effect graph software testing method is still an expensive technique when comparing other black box software testing methods.

Related Works

Many software testing tools have developed to generate and execute test cases. Most of the tools try to automate or semi-automate test case generation and execution processes. Mustafa (Mustafa et al., 2009) analyzes and classifies 135 software testing tools. Mustafa (Mustafa et al., 2009) also denotes that many of available software testing tools are designed for web applications. A comprehensive survey related to automated software test case generation techniques are described in (Anand et al., 2013). Prominent software testing tools that use model-based, combinatorial, symbolic execution, search-based, and adaptive random testing are also denoted (Anand et al., 2013). Another survey (Vishawjyoti et al., 2016) explains automated software test case generation methods and algorithms. Search-based, model based and adaptive random test case generation algorithms are explained in (Vishawjyoti et al., 2016).

Automated cause-effect graph software testing tools are not widely used like other automated software test case generation tools. So, there are an only small number of available tools on the market for cause-effect graph software testing. BenderRBT (Bender, 2008) is one of the commercial tools which was designed to automate test case generation with the cause-effect graphing software testing method. The tool does not only optimize software tests but also provide observability of defects. The tool is recommended for safety critical, mission critical and business critical systems (Bender, 2008).

The Cause-Effect Graph Software Testing

The cause-effect graph software testing is a test case generation method which uses the simplified digital-logic circuit. This method originated from hardware engineering; however, it is adapted to software engineering (Mayers, 2004). The cause-effect graph software testing is a black box testing method and input conditions are systematically combined to generate test cases.

In order to generate test cases, input or test conditions must be determined from software specifications first. This process is called test analysis or test basis. Then, these test conditions are converted to test cases. Determining test conditions is not a complex process and can be done by just reading the software specification document. Everything in a software specification document can be a test condition. Furthermore, some conditions can be derived from experience, which is not documented anywhere. Test conditions may include a large range of possibilities and may not include exact information. However, when generating test cases, the exact software test data must be known. Test cases are derived based on software testing methods. Different software testing methods can derive different test cases from the same test conditions.

Many traditional black box software testing methods such as equivalence partitioning and boundary value analysis do not consider the combination of test conditions. Only single input condition may not cause a failure in the system, however, when two or more input conditions are combined, they may cause a failure (Mayers, 2004). For this reason, each combination of test conditions should be considered when generating test cases. However, even for small and simple software, there is a significant number of combinations of test conditions. Thus, for complex and large projects, the number of combinations becomes astronomical. The cause-effect graph software testing method provides a systematic way to combine test conditions and produce test cases from the combination of test conditions.

(Mayers, 2004) defines six processes that the cause-effect graph software testing involves when generating test cases from software specifications. Figure 1 denotes these processes.

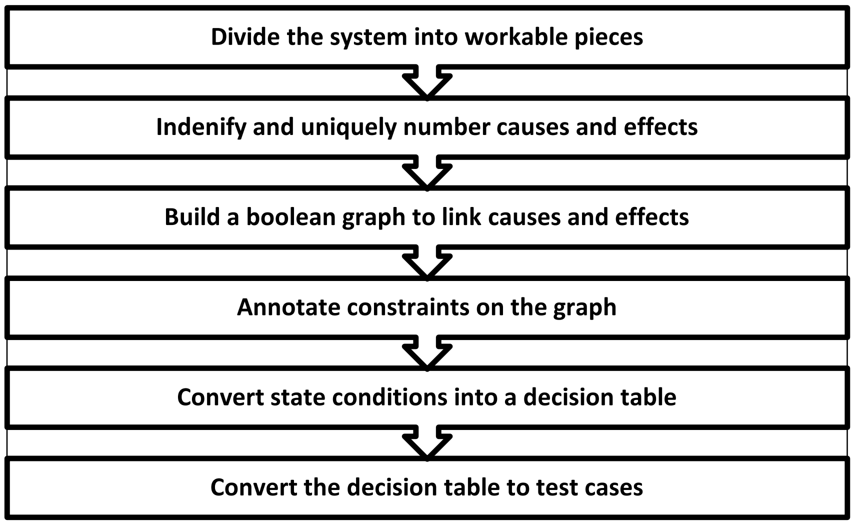

Divide the system into workable pieces
Indenify and uniquely number causes and effects
Build a boolean graph to link causes and effects
Annotate constraints on the graph
Convert state conditions into a decision table
Convert the decision table to test cases

Figure 1. The guideline for the cause-effect graph software testing method

In the first process, the system is decomposed into logical sub-systems. The reason is applying the cause-effect graph software testing to a whole system is cumbersome and not easily manageable. Then each logical subsystem is decomposed into functions, and it is also possible to decompose functions into sub-functions. The second process is identifying causes and effects. In this context, causes refer to input and effects refer to outputs. The causes may be distinct input conditions and equivalence classes, and the effects may be output conditions and system transformations. Both causes and effects are identified from given software specifications. Each keyword or phrase in the software specifications can be a cause. Causes can include hardware events, API calls as well as return codes. In contrast, the effects can include output conditions of equivalence classes, and interaction dialogs and messages. Moreover, all outputs which are generated by a program are considered as an effect. After defining causes and effects, a unique number or name should be assigned to each cause and effect. In addition, it is important to note that the assignment of a unique number or name should be done in such a way that causes and effects can be differentiable. Because both causes and effects are depicted by the same node shape in a cause-effect graph, the only way to differentiate between causes and effects is their unique IDs or names.

After determining causes and effects, they are depicted as nodes in a cause-effect graph. Connections between causes and effects nodes are identified by analyzing the semantic content of software specifications. In this process, middle nodes can be generated to represent combinations of causes. Furthermore, the logic of combination of causes determines the type of middle nodes. Middle nodes are usually ‘AND’ and ‘OR’ Boolean logics. However, all Boolean logics can be applied to causes. Besides, other Boolean logics can be used by constraints. Middle nodes do not only connect causes, but also connect other middle nodes. Figure 2 shows simple notations that are used to draw cause-effect graphs.

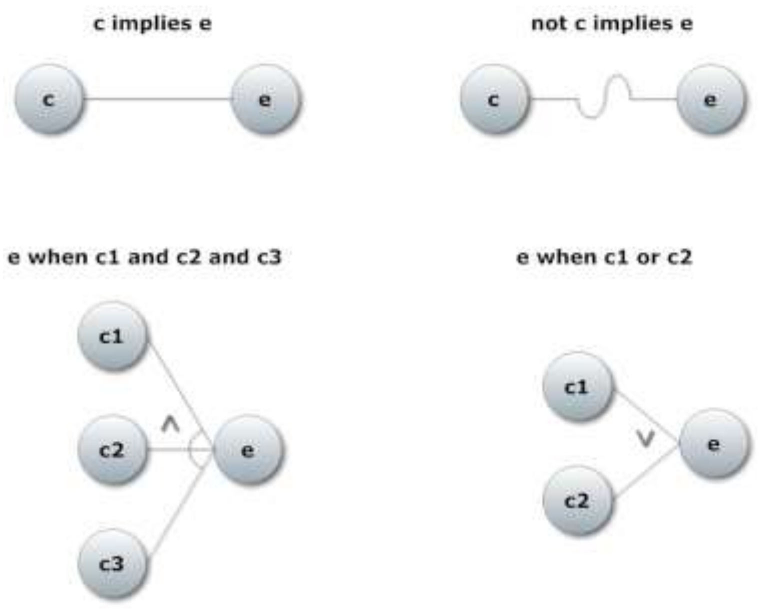


Figure 2. Notation of the cause-effect graph (Mayers, 2004)

The next process is specifying constraints on nodes. Some causes cannot simultaneously be true or at least or at most one node can be true at a time. Moreover, some nodes can mask or require other nodes. Thus, all these constraints must be defined in a cause-effect graph. Five common constraints are defined for cause-effect graphs (Mayers, 2004). Four of them are related to causes and one of them can only be applied to effects.‘E’, ‘I’, ‘O’ and ‘R’ constraints can only be applied to cause nodes. The ‘E’ constraint stands for *Exclusive*. This constraint states that at most one of the nodes can be true. Moreover, all nodes can be false, if only ‘E’ constraint is applied. The ‘I’ constraint stands for *Inclusive*, and it shows that at most one of the nodes can be true. This means that all nodes cannot be false simultaneously, if only ‘I’ constraint is applied. One and only one constraint is represented as ‘O’. This constraint indicates that one and only one of the nodes is true. The last constraint for causes is ‘R’ constraint, which stands for *Requires*. This constraint covers only one node at a time. This means if the node ‘a’ requires the node “b”, then the node ‘b’ will be true whenever the node ‘a’ is true. In other words, if the node ‘a’ is true, the node ‘b’ cannot be false. ‘M’ constraint can only be applied to causes, and it stands for *Mask*. Similar to ‘R’ constraint, it affects one node at a time. If the node ‘a’ masks the node ‘b’, whenever the node ‘a’ is true, the node ‘b’ is forced to be false. Figure 3 shows the representation of all constraints on a cause-effect graph.

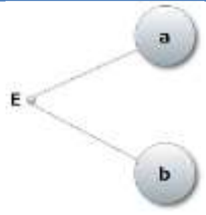
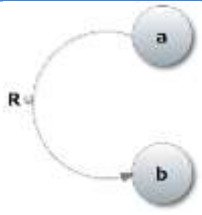
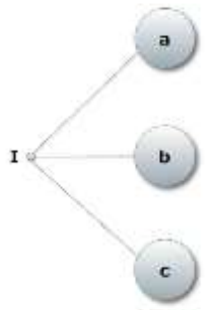
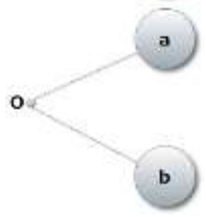
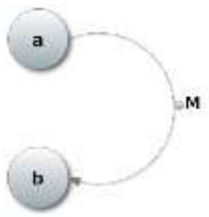
Constraint	Interpretation	Constraint	Interpretation
	‘E’ constraint stands for the exclusive constraint. In this constraint, at most one of these nodes can be true. This means node ‘a’ and node ‘b’ cannot be true simultaneously.		‘R’ constraint stands for requires. If cause ‘a’ is true, then cause ‘b’ must be true. If node ‘a’ is true, node ‘b’ cannot be false
	‘I’ constraint stands for inclusive. In this constraint, at least one of causes is true. This means that all causes cannot be false simultaneously.		
	‘O’ constraint stands for one and only one. If node ‘a’ is true, then node ‘b’ cannot be true.		‘M’ constraint is for effect nodes. ‘M’ stands for mask. If node ‘a’ is true, then node ‘b’ must be false.

Figure 3 – Cause-effect graph constraints and their interpretations (Mayers, 2004)

In the next process, cause-effect graphs are converted to a limited entry decision table. This is also a heuristic process. In the decision table, all causes are located in the first column. Each other columns represents a test case. In order to fill a column, an effect node is selected. Then, all nodes which make it true are identified. If one of these nodes is a middle node, this process is repeated for that node until reaching a cause node. Each OR middle node causes a new row as well as a new test case, and each AND node accumulates previous ANDed conditions. In each iteration, ANDed conditions are combined with current conditions. When filling a column, each row represents a cause; all causes which make it true will be 1. This process is repeated for all effect nodes.

After the conversion, constraints are applied to the decision table. For ‘E’ constraint, if one term is 1, then other terms must be 0. Also, if more than one term is 1 in a column, this column must be discarded because this test case is impossible to generate. Moreover, if all terms in ‘E’ constraint are empty or 0, no additional process is required. For ‘I’ constraint, if all terms are 0 or empty in a column, this column must also be discarded. Otherwise, no additional process is needed. For ‘O’ constraint, if one term is true, the other terms must be 0. If more than one term is true, then this column must be discarded. For ‘R’ constraint, if one term is true and a required term is false, then this column must also be discarded.

A Sample Test Case Generation with the Cause-Effect Graph Software Testing

In this section, a small example which shows the test case generation by the cause-effect graph software testing is explained. In this example, a subpart of the course registration system which assigns course buildings based on the faculty and the number of registered students is specified. Based on the given specification, there are two faculties, which are Engineering Faculty and Art and Science Faculty. Also, there are four buildings, which are A, B, C and D. In addition to that, the following software specifications are given.

- **R0101** If the number of registered students for a course is less than 10 in the Engineering Faculty, the course building will be A block
- **R0102** If the number of registered students for a course is between 10 and 50 in the Engineering Faculty, the course building will be B block
- **R0103** If the number of registered students for a course is less than 10 in the Art and Science Faculty, the course building will be B block
- **R0104** If the number of registered students for a course is between 10 and 50 in the Art and Science Faculty, the course building will be C block
- **R0105** If the number of registered students for a course is greater than 50 both for the Engineering Faculty and the Art and Science Faculty, the course building will be D block

From given above software specifications, causes (input conditions) and effects (outputs) are determined in Table 1. A unique number or ID is assigned for each cause and effect to differentiate nodes in the cause effect graph. Two counters may be used to provide a unique number for each cause and effect. In Table 1, ID of causes start from one and ID of effects starts from one hundred.

Table 1. Determining causes and effects

Causes (input conditions)	**Effects (outputs)**
1. Faculty of Engineering	100. Building is A block
2. Faculty of Art and Science	101. Building is B block
3. Number of registered students < 10	102. Building is C block
4. Number of registered students >= 10 and <50	103. Building is D block
5. Number of registered students > 50	

After determining causes and effects, the cause-effect graph can be drawn by using the Boolean logic that is described in the previous section. Figure 4 shows the cause-effect graph which is corresponding to defined causes and effects in Table 1.

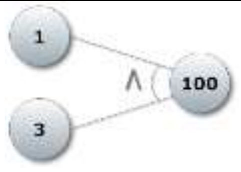
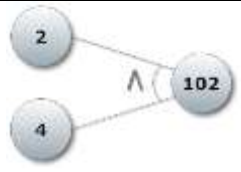
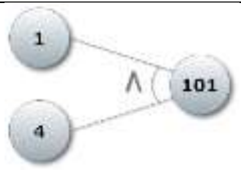
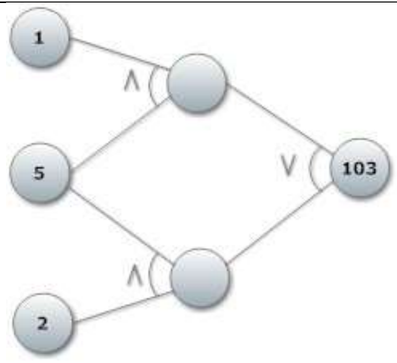
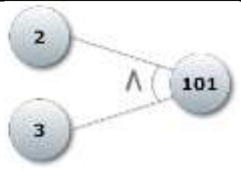
	**Cause:** 1. Faculty of Engineering **AND (∧)** 3. Number of registered students < 10 **Effect:** 100. Building is A block		**Cause:** 2. Faculty of Art and Science **AND (∧)** 4. Number of registered students >=10 and <50 Effect: 102. Building is C block
	**Cause:** 1. Faculty of Engineering **AND (∧)** 4. Number of registered students >=10 and <50 Effect: 101. Building is B block		**Cause:** 1. Faculty of Engineering **AND** (∧) 5. Number of registered students > 50 **OR (V)** 2. Faculty of Art and Science **AND** (∧) 5. Number of registered students > 50 **Effect:** 103. Building is D block
	**Cause:** 2. Faculty of Art and Science **AND (∧)** 3. Number of registered students < 10 **Effect:** 101. Building is B block		

Figure 4. Cause-effect graph of the university course registration system

A single constraint which a faculty cannot be both the Engineering Faculty and the Art and Science Faculty at the same time can be derived from given specifications. For this reason, 'E' constraint can be applied to node1 and node2 in the university course registration system. This constraint is represented in Figure 5.

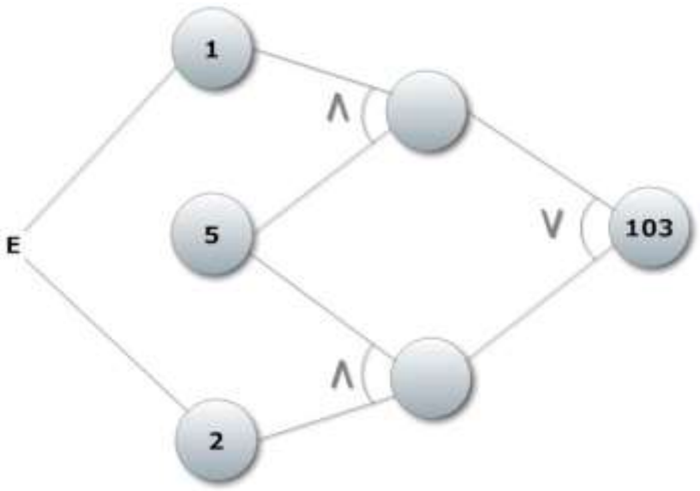


Figure 5. 'E' constraint on causes 1 and 2 in the university course registration system

The next process is converting cause-effect graphs to a decision table. Table 2 represents the decision table of the university course registration system which is obtained after the conversion of cause-effect graphs. For example, in Table 2, the first test case is generated from the first column. In the first column, only "faculty of engineering" and "number of students is less than ten" rows are true in the cause section, and only "A block" is true in the effect section. Thus, the first test case covers the input conditions which is "when the number of students is less than ten" and "the faculty is engineering". The assigned block will be "A block" as an expected output. This process is repeated for all columns. As a result, the number of test cases will be equal to the number of columns, if no optimizations are applied to the decision table. In that example, six test cases are produced. The test suite is shown in Table 3.

Table 2. The decision table for the university course registration system

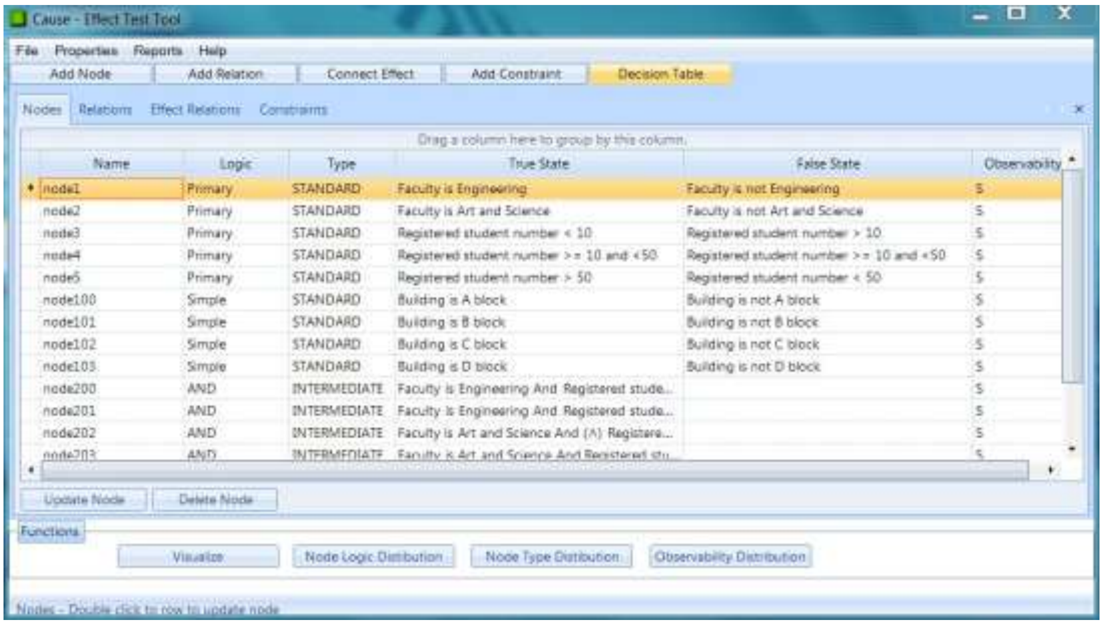
Causes:	**1**	**2**	**3**	**4**	**5**	**6**
1. Engineering	1	1	0	0	1	0
2. Art and Science	0	0	1	1	0	1
3. < 10	1	0	1	0	0	0
4. >= 10 and <50	0	1	0	1	0	0
5. > 50	0	0	0	0	1	1
Effects:						
100. A block	1	0	0	0	0	0
101. B block	0	1	1	0	0	0
102. C block	0	0	0	1	0	0
103. D block	0	0	0	0	1	1

Table 3. Test suite for the university course registration system

Test case #	Faculty	# of students	Expected Output
#1	Engineering	5	A block
#2	Engineering	20	B block
#3	Art and Science	4	B block
#4	Art and Science	25	C block
#5	Engineering	55	D block

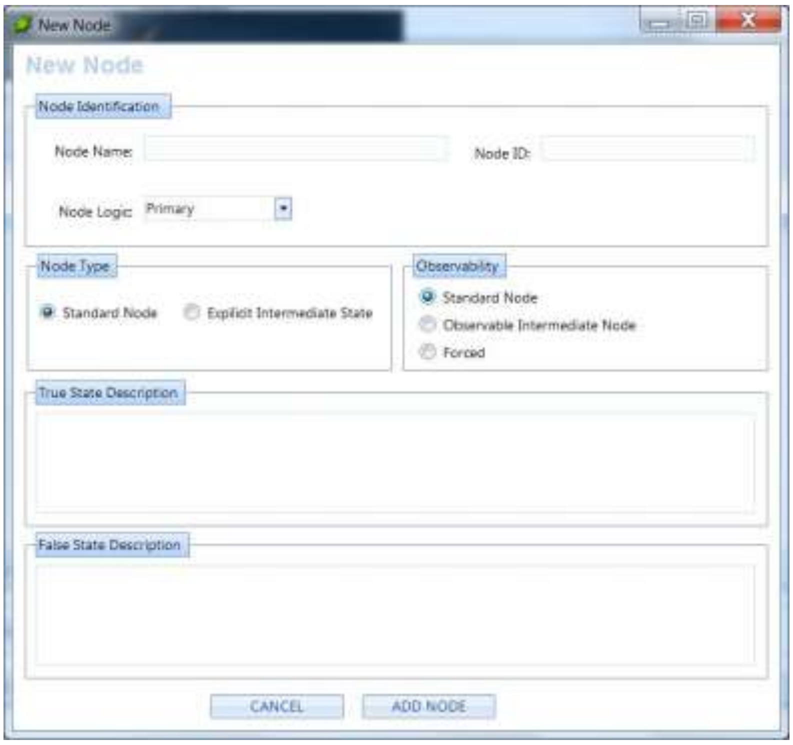
The interface of the cause-effect test tool is depicted in Figure 6. In the cause-effect graph software testing tool, all causes, effects, and relations between nodes should be determined first. All causes, effects and their combinations are defined as a node in the system. In the tool, all types of nodes are defined in the “new node” section, which is depicted in Figure 7.

When defining a new node, a unique name and an ID should be assigned for each node. Then, node logic, type, observability, true and false state description should be provided. The following paragraph explains details of those specifications that need to be defined for each node. (Bender, 2008) also defines similar specifications in their cause-effect graph software testing tool to specify each node.



Name	Logic	Type	True State	False State	Observability
node1	Primary	STANDARD	Faculty is Engineering	Faculty is not Engineering	S
node2	Primary	STANDARD	Faculty is Art and Science	Faculty is not Art and Science	S
node3	Primary	STANDARD	Registered student number < 10	Registered student number > 10	S
node4	Primary	STANDARD	Registered student number >= 10 and <50	Registered student number >= 10 and <50	S
node5	Primary	STANDARD	Registered student number > 50	Registered student number < 50	S
node100	Simple	STANDARD	Building is A block	Building is not A block	S
node101	Simple	STANDARD	Building is B block	Building is not B block	S
node102	Simple	STANDARD	Building is C block	Building is not C block	S
node103	Simple	STANDARD	Building is D block	Building is not D block	S
node200	AND	INTERMEDIATE	Faculty is Engineering And Registered stude...		S
node201	AND	INTERMEDIATE	Faculty is Engineering And Registered stude...		S
node202	AND	INTERMEDIATE	Faculty is Art and Science And (A) Registere...		S
node203	AND	INTERMEDIATE	Faculty is Art and Science And Registered stu...		S

Figure 6. The cause-effect graph software testing tool



Node Logic: Primary

Node Type: [✓ Standard Node] Explicit Intermediate State

Observability: [✓ Standard Node] Observable Intermediate Node Forced

Figure 7. Adding a new node in the cause-effect graph software test tool

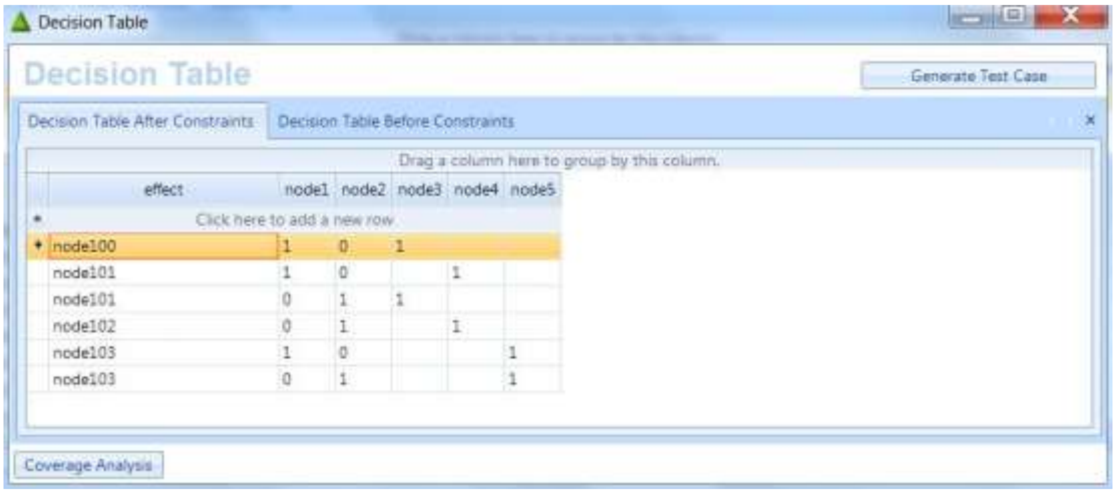
The node logic identifies the type of node. In the tool, causes are primary nodes and effects are simple nodes. Other middle nodes, which are the combination of causes, are defined by Boolean logics such as "AND" or "OR". If a node is either a cause or an effect, the node type will be "Standard". In addition, if a node is a middle node, the node type will be "Explicit Intermediate State". The true state description determines the case when the input condition or the output of the node is true. In the same way, the false state description determines the case when an input condition or an output of the node is false. If a node is an explicit intermediate state, the false state description is usually empty. The observability determines the availability of test state during the execution of test cases. For example, all objects on screens, database transactions, and objects on reports are considered as observable nodes. In addition, sounds and movements can also be considered as additional observable objects. In the context of requirement specification testing, all cause nodes are assumed to be observable and all effect nodes are assumed to be not observable. Thus, during the software testing, if a defect is found in an intermediate node, the cause of the defect can only be found by tracking an observable cause node. In the tool, if a middle node is not observable, the observability must be set as a standard node. However, if a middle node is actually observable, then the observability should be set as "Observable Intermediate Node". Nodes may not be testable in some cases because of given constraints, although the node itself is observable. In that case, the observability should be set to "Forced".

After identifying and defining nodes, the connections of nodes should be identified. In the new tool, there are two types of connections. The first connection type connects cause nodes to middle nodes, or middle nodes to other middle nodes. The second connection type connects cause nodes to effect nodes, or middle nodes to effect nodes. In the tool, the first type of connection can be defined from "Add Relation" section and the second type of connection can be defined from the "Connect Effect" section.

The next process is defining constraints on nodes. In the tool, two types of constraints are defined. The first type of constraints includes "exclusive", "inclusive" and "one and only one" which can be applied to more than two nodes. The second type of constraints covers "requires", and "mask" constraints, which involve only two nodes. Furthermore, the constraint direction determines which node is required or masked. Constraints are defined in "Add Constraint" section in the tool.

After defining all connections, and constraints, the decision table can be generated by using the "Decision Table" module. In the decision table, the first column shows the effect node and other columns represent different "primary" cause nodes. The algorithm of generating decision table starts from "effect" nodes. For each primary node that connects to an "effect" node with "AND" relation is marked "1" and creates a row in the decision table. If the connected node is an intermediate node, the algorithm recursively iterates until reaching a primary node. Each "OR" relation generates *n* number of rows for each connected node in the decision table. After generating the decision table, the constraints on cause and effect nodes are applied. If a constraint contradicts with a condition in a raw, the raw is deleted from the decision table. If a constraint does not conflict with conditions in a raw, some cases may either turn to true or false from "don't care" cases which does not affect outputs. There exist different decision table generation algorithms. (Aditya, 2008) and (Srivastava et al., 2009) cover some algorithms to generate decision table from a cause effect graph. Depends on the selected algorithm, space and time complexity can vary.

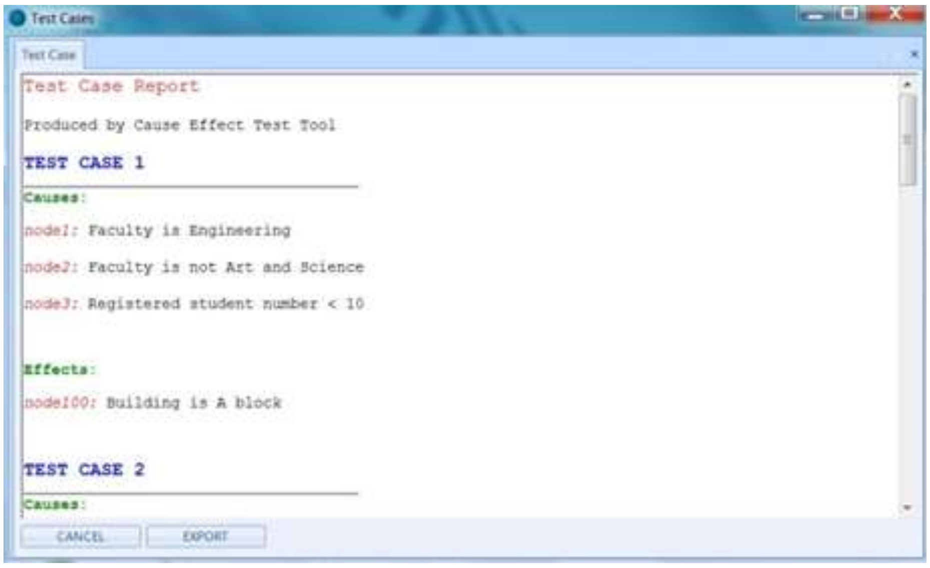
The decision table for the university course registration system is represented in Figure 8. Two decision tables are produced in the tool. The first table represents the decision table which all constraints are applied on it. The second table shows the decision table without constraints. In this way, effects which are canceled by constraints can be detected.


Decision Table
Decision Table
Generate Test Case
Decision Table After Constraints
Decision Table Before Constraints
Drag a column here to group by this column.
Click here to add a new row
Coverage Analysis

effect	node1	node2	node3	node4	node5
node100	1	0	1		
node101	1	0		1	
node101	0	1	1		
node102	0	1		1	
node103	1	0			1
node103	0	1			1

Figure 8. The decision table for the university course registration system after applying constraints on cause and effect nodes.

Test cases can be directly generated from a decision table. Each row constitutes a test case. Test cases can be produced by checking input conditions in a row. For example, for node100 in Figure 8, the node1 and node3 are 1 (true) and node2 is 0 (false). So, the test case will cover the case when statements in node1 and node3 are true, and node2 is false. The expected output will be the case when the statement in node100 is true. In the tool, test cases can be produced by "Generate Test Case" section. Figure 9 shows the test cases which are produced from the decision table in Figure 8.



```
Test Case Report

Produced by Cause Effect Test Tool

TEST CASE 1
Causes:
node1: Faculty is Engineering
node2: Faculty is not Art and Science
node3: Registered student number < 10

Effects:
node100: Building is A block

TEST CASE 2
Causes:
```

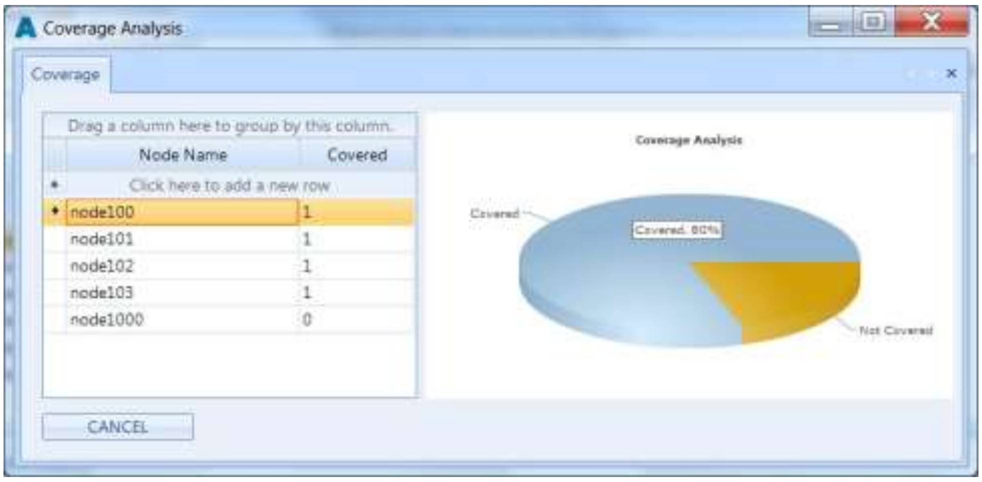
Figure 9. The test suite for the university course registration system in the cause-effect graph software test tool

The Effect Coverage Analysis

After the cause-effect graph software testing, at least one test case is generated for each effect (output). However, some effects may not be covered after generating all test cases because some constraints may cause contradictions with given input conditions or an effect relation may not be defined or deleted by mistake. The effect coverage (EC) can be calculated by the following formula.

$$EC = \frac{Number\ of\ different\ effects\ in\ test\ cases}{Total\ number\ of\ defined\ effects}\ x\ 100$$

One of the goals of cause-effect graph software testing tools is reaching 100% effect coverage so that all effects (outputs) are tested at least once. Figure 10 represents the coverage analysis of the university course registration system when adding an extra effect node (node1000) which has no relation to any causes. Thus, after the node1000, the effect coverage is reduced to 80%.



Node Name	Covered
node100	1
node101	1
node102	1
node103	1
node1000	0

Figure 10. The effect coverage analysis in the cause-effect graph software test tool

Having less than full (100%) coverage does not always indicate a problem. However, uncovered effects should be investigated to find potential problems in software specifications. Coverage analysis can also find some minor mistakes such as missing connections to effect nodes.

CONCLUSION

Software test case generation tools are one of the important tools to automate generation of test cases. The cause-effect graph software testing tool is one of the test generation tools that use software specifications as an input to generate test cases. The cause-effect graph software testing tool provides a systematic way to combine input conditions. The combination of different input conditions may reveal new bugs which may not be found with other software test case generation methods.

In this paper, a new cause-effect graph software testing tool is described. Although the tool automates test case generation from cause-effect graphs, building a cause-effect graph still requires manual processes such as finding causes, effect, and constraints from software specifications.

Cause-effect graph software testing tools are not usually preferred by small sized software projects. Some of the reasons are the cost of processes and insufficient tools. In cause-effect graph software testing tools, all input conditions and outputs should be formally or semi-formally defined in the tool. In additions, all relations between causes and effects should be defined. This process requires huge resources in terms of tester labor. However, cause-effect graph software testing method is just found some part of bugs in the system. For this reason, it must be used complementarily with other testing methods.

REFERENCES

Aditya, P. (2008) Software Testing, Pearson Publication.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., Harman, M., Harrold, M. J. and McMinn, P. (2013) An orchestrated survey of methodologies for automated software test case generation, Journal of Systems and Software, Volume 86, Issue 8, Pages 1978 – 2001.

Bender, R. (2008) Cause-Effect Graphing User Guide (1st ed.), NY: Bender RBT Inc.

Burnstein I. (2002) Practical Software Testing: a process-oriented approach (1st ed.), NY: Springer-Verlag Inc.

Everett, G. D. and McLeod, R. (2007) Software testing: testing across the entire software development life, IEEE Press, USA: A John Wiley & Sons Inc.

Lewis, W. E. (2009) Software Testing and Continuous Quality Improvement. Third Edition, CRC Press.

Mayers, G. J. (2004) The art of software testing (2nd ed.), USA: John Wiley & Sons Inc.

Mustafa, K. M., Al-Qutaish, R.E. and Muhairat, M. I. (2009) Classification of Software Testing Tools Based on the Software Testing Methods, 10.1109/ICCEE.2009.9, Pages: 229 – 233, IEEE Conference Publications.

Nook, R. (2008) Advanced Software Testing Vol. 1, Rock Nook Inc.

Srivastava, P. R., P. Patel and Chatrola, S. (2009) Cause effect graph to decision table generation, ACM SIGSOFT Software Engineering Notes, Volume 34 Issue 2.

Vishawjyoti and Gandhi, P. (2016) A survey on prospects of automated software test case generation methods, 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom) IEEE, Pages: 3867 – 3871.