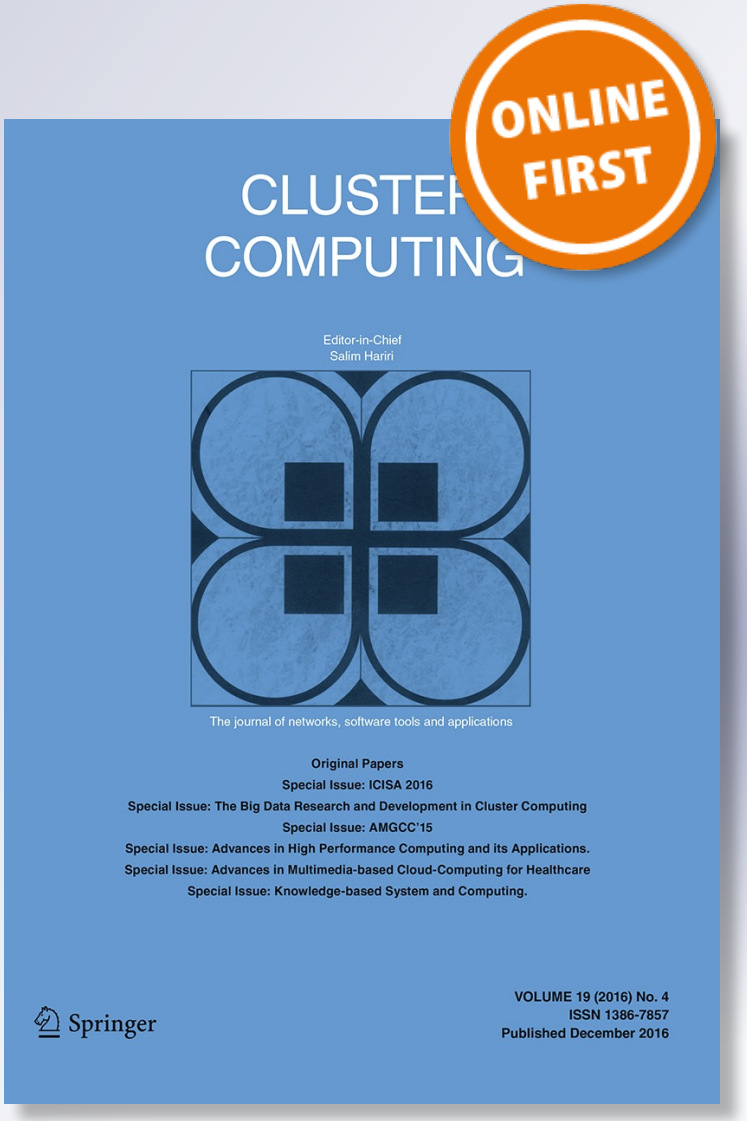
# *MCCFG: an MOF-based multiple condition control flow graph for automatic test case generation*

**Hyun Seung Son, Young B. Park & R. Young Chul Kim**

**Cluster Computing**
The Journal of Networks, Software Tools and Applications

ISSN 1386-7857

Cluster Comput
DOI 10.1007/s10586-016-0687-1




Springer

Cluster Comput
DOI 10.1007/s10586-016-0687-1

# MCCFG: an MOF-based multiple condition control flow graph for automatic test case generation

**Hyun Seung Son**[1] · **Young B. Park**[2] · **R. Young Chul Kim**[1]

Received: 22 March 2016 / Revised: 3 October 2016 / Accepted: 18 November 2016


**Abstract** Requirement-based testing (RBT) is widely known for the efficient testing in the limited resources. However RBT is difficult to generate automatic test cases; thus it needs complex methods. This paper suggests our automatic test case generation for all coverage (statement, condition, decision, condition/decision, modified condition/decision, and multiple condition coverage) based on the model-based testing. To do this, we extend the original control flow graph with multiple conditions for all *condition related coverage, which is called multiple conditions control flow graph*, and adapt a model transformation using metamodel mechanism for test case generation. As a result, our proposed method successfully applies to the prior test requirement.

**Keywords** Automatic test case generation · Control flow graph (CFG) · Coverage-based testing · Multiple condition control flow graph (MCCFG)

## 1 Introduction

The recent software industries start to recognize the importance of the software quality. While considering this fact, the software quality is also critical in the embedded software system. Even the small malfunction of software can significantly damage the entire system [1]. Therefore, the software testing can be used to improve the software quality [2]. Especially, for the large scale and complex software with limited resources, it is required to have much more efficient testing method.

However, the complete testing is impossible in any software fields due to the necessity of infinite input values, paths, and timings [3–5]. Hence, in most of software except the safety critical system, it does not aim at complete testing [6]; instead, most software tries to test as much as possible. In this issue, automatic tool with diverse testing techniques is efficient for test case generation [3].

On the other hand, the risk-based testing [7] needs the one that can identify the possible, potential problems for the software development. While using the risk-based testing, we also apply prior requirement with all coverage, implementing the automatic test case generation for all different test coverages. Therefore, we focus on different levels of coverage according to the priority of software, which can be stated as the following order: statement coverage (SC) < condition coverage (CC) < decision coverage (DC) < condition/decision coverage (CDC) < modified condition/decision coverage (MCDC) < multiple condition coverage (MCC) [8]. To generate the test case based on the various coverage, we use control flow graph (CFG) which expressed as multiple nodes and paths between an initial node and a final node. This is a very popular method for the test case generation. However, by using CFG, the flow paths are clearly represented during the program execution. Since this original CFG expresses the control flow, a test case is only generated based on SC and DC for the coverage. In other words, the CFG cannot generate test case in multiple conditions related coverage such as CDC and MCDC. Hence, in order to solve this prob-

✉ R. Young Chul Kim
bob@selab.hongik.ac.kr

Hyun Seung Son
son@selab.hongik.ac.kr

Young B. Park
ybpark@dankook.ac.kr

[1] Department of Computer and Information Communication, Hongik University, Sejong 30016, Korea

[2] Department of Computer Science and Engineering, Dankook University, Yongin 16890, Korea

Published online: 09 December 2016

lem, we propose the multiple conditions CFG (MCCFG) for dealing with multiple conditions.

The proposed MCCFG can express either multiple conditions for a simple unit or combined unit with notation of "AND" or "OR". For automation issue, we use metamodel mechanism in order to design metamodel of MCCFG based on Meta object facility (MOF) [9], and develop graphical notation of MCCFG.

In this paper, we propose the automatic test case generation method to generate an automatic test case based on the coverage from MCCFG. The method consists of branch extraction and condition extraction. While the branch extraction is to generate the test case with branch based on SC or DC, the condition extraction is to generate test case by condition based on CDC and MCDC. In order to generate the test case automatically, we design metamodel of MCCFG and implement the automatic tool based on Eclipse Modeling Framework (EMF) [10]. This paper shows the graphical notation of MCCFG and generates test case from MCCFG. With a case study, we demonstrate how to generate the test cases by four test coverage cases of SC, DC, CDC, and MCDC.

The paper mentions in following order: Sect. 2 describes related works. Section 3 shows metamodel and graphical notations of MCCFG. Section 4 mentions our test case generation from MCCFG based on the coverage criteria. Section 5 shows case studies. Section 6 gives a conclusion and future works.

## 2 Related works

Control Flow Graph (CFG) is used to express all the different control flows in a computer program. CFG uses a data structure for code optimization in compiler. After this, CFG applies to extend in software engineering and software testing [11]. CFG is also a model expressed as multiple nodes and paths. It shows other paths between initial node and final node, that is, called control flow paths. CFG shows other paths during program execution.

CFG consists of code-based CFG (CBCFG) and model-based CFG (MBCFG), which separates based on the selection from source code information. CBCFG is a white box testing technique based on a source code, but widely used in software testing area. MBCFG is a black box testing technique based on a model with information of control flow. CFG is useful tool to generate test case on coverage [12].

CFG has the coverage as follows [13]:

- Statement Coverage (SC): All nodes in the CFG must visit at least once.
- Decision Coverage (DC): Decision points that perform the true/false in all edges of the CFG are visited at least once.
- Path Coverage (PC): All paths in the CFG must be covered.

CFG which cannot generate test case about conditions is as follows [12]:

- Condition Coverage (CC): the condition at least once in either true or false must be coverage.
- Condition/Decision coverage (CDC): each condition and decision point with either true or false must be coverage at least once.
- Modified Condition/Decision coverage (MCDC): the result of each condition and the entire condition must be coverage.
- Multiple Condition Coverage (MCC): all possible combinations on each condition must be coverage.

In the model-based CFG (MBCFG) techniques, we select a few existing research [14–16] and compare this work according to six criteria as shown Table 1. All of existing techniques are based on UML Sequence Diagram (SD). Only one of this work propose both UML and code. Some of the existing CFGs are Inter-procedural Restricted Control Flow Graph (IRCFG), Petri-Net, and Concurrent Control Flow Graph (CFG). But, all of existing CFGs does not allow the multi coverage.

## 3 Our proposed multiple condition control flow graph (MCCFG)

The proposed method generates test case from MCCFG based on each coverage in Fig. 1. This method suggests adopting a model transformation for test case generation. The model transformation technique is to automatically transform model to model based on metamodel. Therefore, we define graphical notation of MCCFG, and make a metamodel related to MCCFG. Also define rules of the test case generation through EMF. Finally we can generate test cases with metamodel of MCCFG based on coverage criteria. Figure 1 shows the whole structure for multiple test case generation.

### 3.1 The Metamodel of MCCFG

In order to define the metamodel of MCCFG, we refer the metamodel of UML activity diagram. "These (UML Activities) are commonly called control flow and object flow models", which is written in UML 2.4 specification [17]. In short, UML activity diagram may be used to CFG. Therefore, we refer the metamodel of UML activity diagram because the metamodel in UML 2.4 specification is complicated. Last but not least we redesign the metamodel of MCCFG in order to add elements of multiple condition like Fig. 2. The root element named *MCCFGModel* consists of Group that has multiple choices. This group consists of *NodeElement* and

**Table 1** Comparison of existing MBCFG techniques

| | [14] | [15] | [16] | This work |
|---|---|---|---|---|
| 1. Source information | SD | SD | SD | UML and Code |
| 2. UML version | UML 1.x | UML 1.x | UML 2.0 | UML 2.4 |
| 3. Produced CFG | IRCFG | Petri-Net | CCFG | MCCFG |
| 4. Loop | No | No | Yes | Yes |
| 5. Condition | Yes | No | Yes | Yes |
| 6. Multi coverage | No | No | No | Yes |

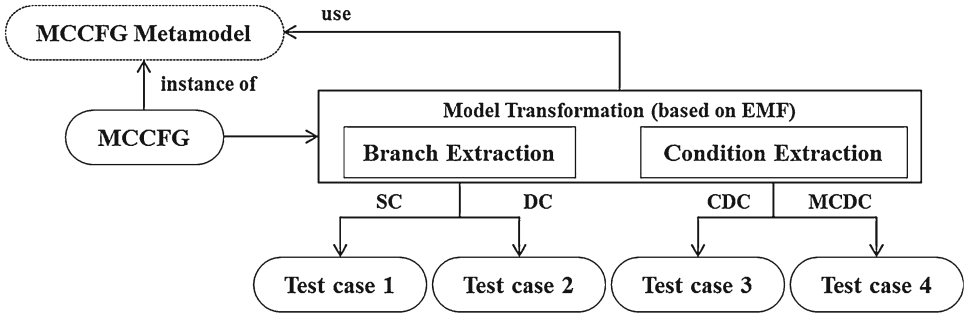



**Fig. 1** The whole structure for multiple test case generation

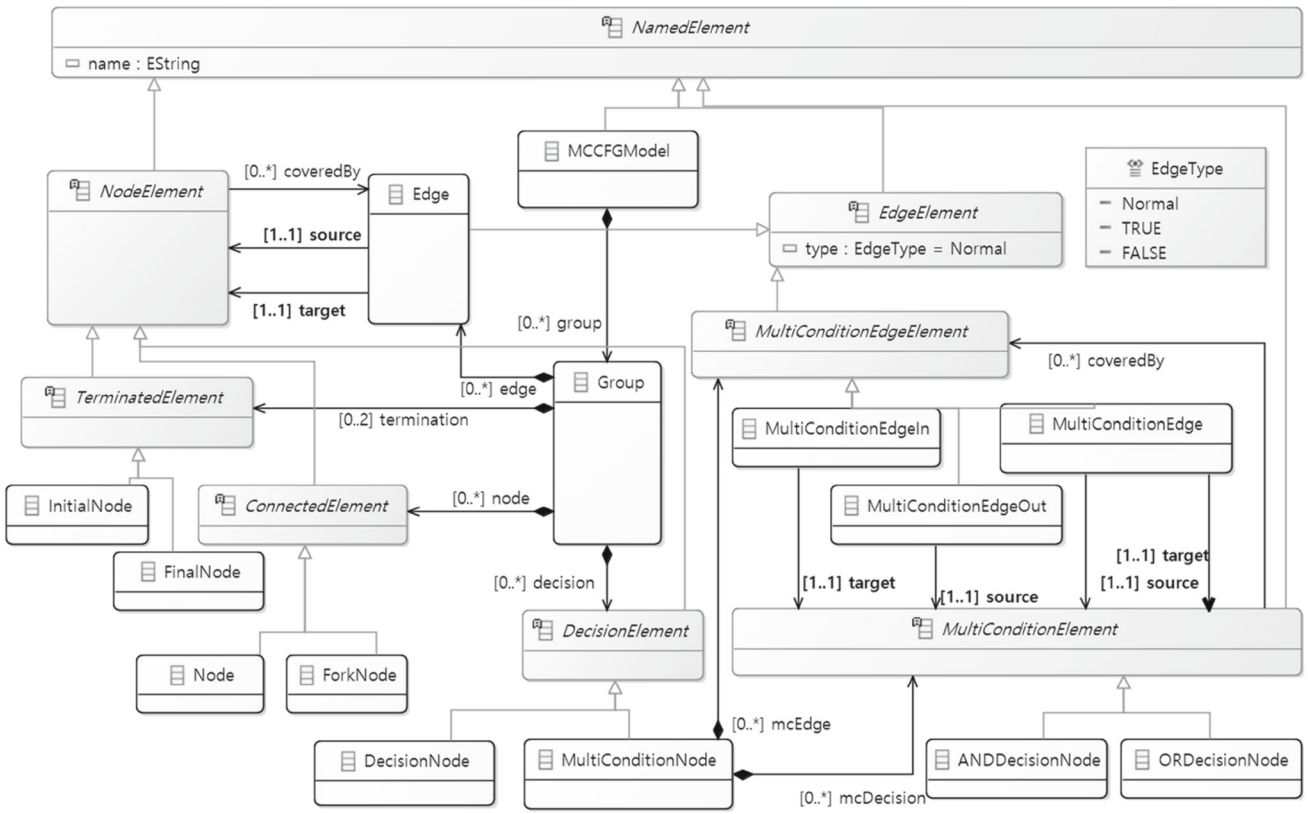



**Fig. 2** The metamodel of MCCFG

**Table 2** Graphical notations of MCCFG

| Type | Notation | Comment |
|---|---|---|
| Initial Node | | Initial node of all nodes |
| Final node | | Final node of all nodes |
| Node | name | One node |
| Fork node | | Concurrent node |
| Decision node | | Decision according to condition |
| Multi condition node | | Represents multiple conditions |
| OR decision node | | AND conditional expression is described |
| AND decision node | | OR conditional expression is described |
| Group | | To tie nodes and edges is described |
| Edge | | Connection of node and node |
| Multi condition edge (blank or in or out) | | Connection between multiple conditions is described |

Edge. *NodeElement* is an abstract node that includes *InitialNode*, *FinalNode*, *Node*, *ForkNode*, *DecisionNode*, and *MultiConditionNode*. Edge is a bridge to connect node and node. Figure 2 shows the metamodel of MCCFG.

The big difference between the original CFG and the MCCFG is *MultiConditionElement* that consists of *ORDecisionNode* expressed "OR" notation and *ANDDecisionNode* expressed "AND" notation. In order to separate the paths and conditions, we design including *MultiCondtionNode* inside the condition that relates the node and edge.

### 3.2 Graphical notation of MCCFG

The graphical notation of MCCFG consists of *Initial Node, Final Node, Node, Fork Node, Decision Node, Multi Condition Node, OR Decision Node, And Decision Node, Group, Edge,* and *Multi Condition Edge* in table 2.

Figure 3 shows one example of metamodel in MCCFG. The conditional expression of 'Multi Condition Element In' is one case of "A and (B or C)". It is described this condition according to priority. "OR" conditional expression is located

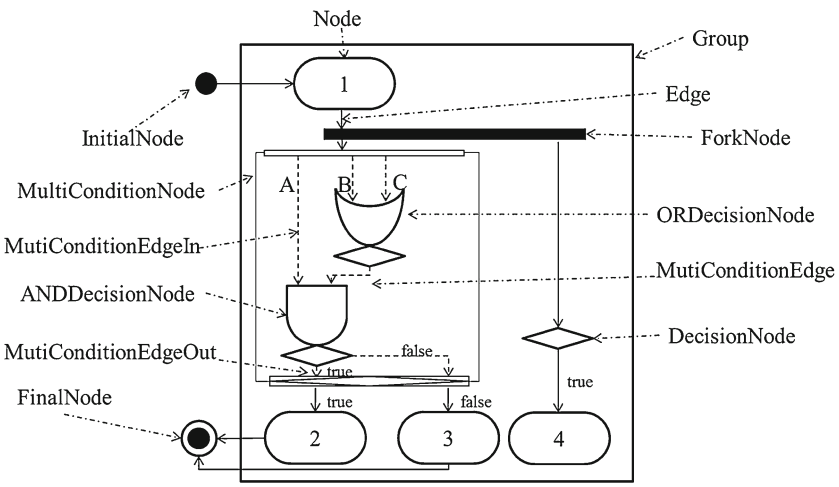



**Fig. 3** Example of modeling using MCCFG

**Fig. 4** XML of modeling using MCCFG

```
<?xml version="1.0" encoding="UTF-8"?>
<MCCFGModel:MCCFGModel xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:MCCFGModel="http://selab.hongik.ac.kr/MCCFGModel">
  <group>
    <decision xsi:type="MCCFGModel:MultiConditionNode">
      <mcEdge xsi:type="MCCFGModel:MultiConditionEdgeIn" name="A"
target="//@group.0/@decision.0/@mcDecision.0"/>
      <mcEdge xsi:type="MCCFGModel:MultiConditionEdgeIn" name="B"
target="//@group.0/@decision.0/@mcDecision.1"/>
      <mcEdge xsi:type="MCCFGModel:MultiConditionEdgeIn" name="C"
target="//@group.0/@decision.0/@mcDecision.1"/>
      <mcEdge xsi:type="MCCFGModel:MultiConditionEdge"
source="//@group.0/@decision.0/@mcDecision.1" target="//@group.0/@decision.0/@mcDecision.0"/>
      <mcEdge xsi:type="MCCFGModel:MultiConditionEdgeOut" type="TRUE"
source="//@group.0/@decision.0/@mcDecision.0"/>
      <mcEdge xsi:type="MCCFGModel:MultiConditionEdgeOut" type="FALSE"
source="//@group.0/@decision.0/@mcDecision.0"/>
      <mcDecision xsi:type="MCCFGModel:ANDDecisionNode" name=""/>
      <mcDecision xsi:type="MCCFGModel:ORDecisionNode"/>
    </decision>
    <decision xsi:type="MCCFGModel:DecisionNode"/>
    <node xsi:type="MCCFGModel:Node" name="1"/>
    <node xsi:type="MCCFGModel:ForkNode"/>
    <node xsi:type="MCCFGModel:Node" name="2"/>
    <node xsi:type="MCCFGModel:Node" name="3"/>
    <node xsi:type="MCCFGModel:Node" name="4"/>
    <termination xsi:type="MCCFGModel:InitialNode"/>
    <termination xsi:type="MCCFGModel:FinalNode"/>
    <edge source="//@group.0/@termination.0" target="//@group.0/@node.0"/>
    <edge source="//@group.0/@node.0" target="//@group.0/@node.1"/>
    <edge source="//@group.0/@node.1" target="//@group.0/@decision.0"/>
    <edge source="//@group.0/@node.1" target="//@group.0/@decision.1"/>
    <edge type="TRUE" source="//@group.0/@decision.0" target="//@group.0/@node.2"/>
    <edge type="FALSE" source="//@group.0/@decision.0" target="//@group.0/@node.3"/>
    <edge type="TRUE" source="//@group.0/@decision.1" target="//@group.0/@node.4"/>
    <edge source="//@group.0/@node.2" target="//@group.0/@termination.1"/>
    <edge source="//@group.0/@node.3" target="//@group.0/@termination.1"/>
  </group>
</MCCFGModel:MCCFGModel>
```

first because of the priority. “AND” conditional expression is located later. Especially, “OR” and “AND” decision nodes is only available to the two inputs.

Figure 4 shows a XML code of metamodel in MCCFG. The notations of MCCFG is expressed to metamodel’s each elements from a source model. The XML code is generated by automatic tools.

On the CFG, it expresses only a decision node without giving a reference to the number of conditions and existing problems. Therefore, our idea can describe conditions in both models for CDC or MCDC. The proposed MCCFG can be solved in this problem, which can possibly generate the test case based on MCDC.

This paper just emphasizes on Step 3, Step 4, and Step 5 within the proposed test process. Figure 5 shows the possible combination of conditional expression in MCCFG. We specially limit to two input values for unification on all conditions. The unification of conditions expresses as follows: (1) is “AND” and (2) is “OR”. The multiple conditions are expressed as follows: (3), (4), (5), and (6). In these expression, MCCFG generates the test case between low-level and high-level with the branch as well as the condition.

## 4 Test case generation and execution

MCCFG is designed on MOF. We can develop the automatic transformation tool for test case generation using EMF. Figure 6 shows several steps for the test case generation. This method separates two steps such as branch extraction and condition extraction from MCCFG. The branch extraction is used to generate test case for SC and DC based on paths. This method executes the process as follows: (1) to generate all paths from MCCFG, (2) to merge decision and condition nodes, (3) to extract nodes. The condition extraction is used to generate the test case for CDC and MCDC based on conditions. This method executes the process as follows: (1) to translate condition node, (2) to generate combination of condition data.

### 4.1 Branch extraction

Figure 7 shows the definition of MCCFG model. The relationship of MCCFG between the nodes and the edges is defined. The notation defines four tuple $\{N, E, T, n_0\}$ where N is a set of nodes, E is a set of edges, T is transition, $n_0$ is ini-

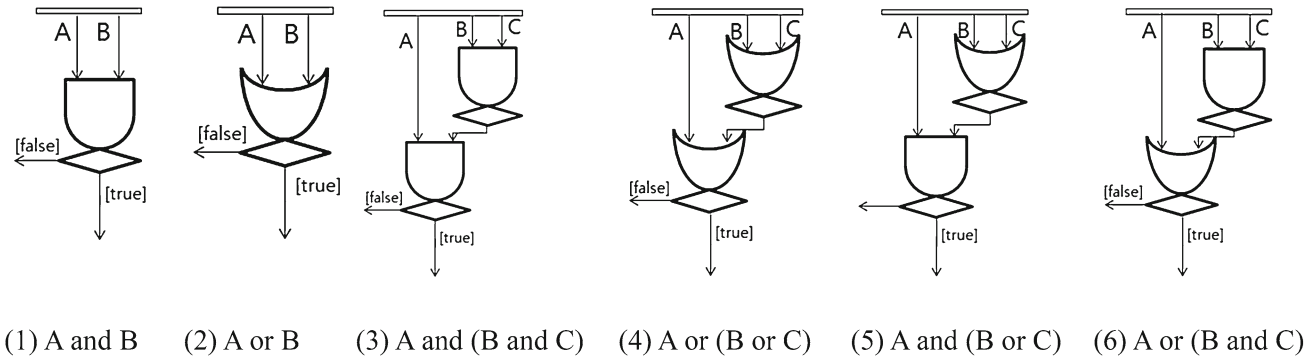



**Fig. 5** Possible conditional expression in MCCFG

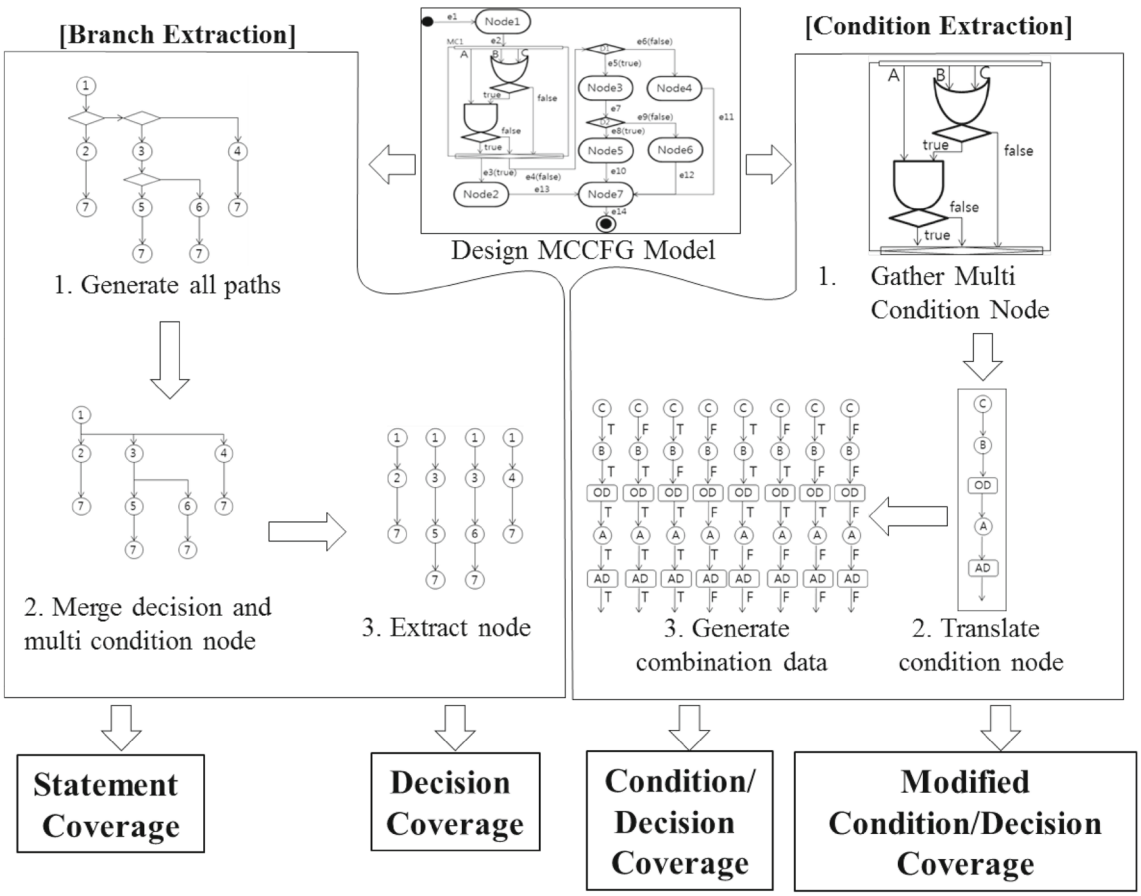



**Fig. 6** Test case generation method from MCCFG

tial node. This definition of notation is used when generating all paths.

Figure 8 shows the algorithm of all path generations from MCCFG. This algorithm, which is similar to depth first search (DFS), finds all paths from MCCFG (that is, on all paths with decision node and multiple condition node). In addition, the algorithm extracts a node from the merged paths. From the founded nodes, it can easily generate test case. 1) SC executes all nodes at least once. So, if we can print out with excluding the marked nodes, then they can generate test cases. 2) DC is decision points to perform the true/false at least once; all branch nodes are generated as all test cases.

### 4.2 Condition extraction

In order to process condition extraction, we preferentially collect MultiConditionNode from MCCFG. Because Multi-ConditionNode is separated into metamodel level, we easily collect information.

Figure 9 shows multiple condition data generation. We can extract the combination data from MultiConditionNode. MultiConditionNode is difficult to express the relationship between input value and branch. To effectively handle the relationship, we propose a model named *condition node* in Fig. 9. The condition node serializes inside nodes of MultiCondtionNode. This condition node describes a logic expression. We can make the result of output condition value after entering the input condition value of all possible combination through calculation. This output condition values are used when generating the test case based on CDC and MCDC.

Figure 10 shows virtual table for test case generation based on CDC. With the result in the step of multiple conditions for all combination of conditions, we must choice the condition

MCCFG = <N, E, T, $n_0$>

– N : a finite set of nodes

– E : the finite set of transition labels

– T $\subseteq$ N x E x N : the set of transitions

– $n_0$ : the initial node of the MCCFG

**Fig. 7** The definition of MCCFG model

| | T | F |
|---|---|---|
| 1 | TT | FF |

(1) A and B

| | T | F |
|---|---|---|
| 1 | TT | FF |

(2) A or B

| | T | F |
|---|---|---|
| 1 | TTT | FFF |

(3) A and (B and C)

| | T | F |
|---|---|---|
| 1 | TTT | FFF |

(4) A or (B or C)

| | T | F |
|---|---|---|
| 1 | TTT | FFF |

(5) A and (B or C)

| | T | F |
|---|---|---|
| 1 | TTT | FFF |

(6) A or (B and C)

**Fig. 10** Virtual table for test case generation based on CDC

```
MCCFGModel rootModel = (MCCFGModel) resource.getContents().get(0);
Iterator<Group> iGroup = rootModel.getGroup().iterator();
while(iGroup.hasNext()) {
    Group group = iGroup.next();
    NodeElement initE = FindInitialNode(group);
    NodeElement finalE = FindFianlNode(group);
    String str = "";
    DepthSearch(initE,initE,finalE,str);
}
DepthSearch(NodeElement initialNode, NodeElement node, NodeElement
finalNode, String out) {
    if( !node.equals(finalNode) ) {
        List nextNodeList = new ArrayList();
        NextNode(nextNodeList,node);
        Iterator<EventActionObject> iNextNode=nextNodeList.iterator();
        while(iNextNode.hasNext()) {
             EventActionObject eao = iNextNode.next();
             out += "("+eao.getCurNode().getName()+",";
             out += eao.getEvent().getName()+",";
             out += eao.getNextNode().getName()+")";
             DepthSearch(initialNode, eao.getNextNode(),finalNode,out);
        }
    }
}
```

**Fig. 8** The algorithm of all paths generation from MCCFG

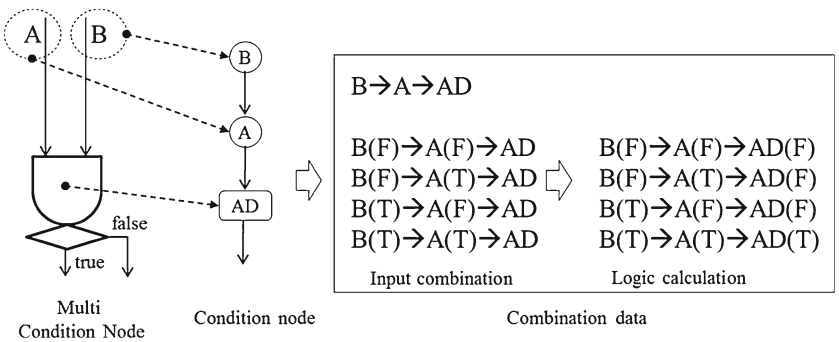



**Fig. 9** Multiple condition data generation

| | T | F |
|---|---|---|
| 1 | TT | FT |
| 2 | ~~TT~~ | TF |

(1) A and B

| | T | F |
|---|---|---|
| 1 | TF | FF |
| 2 | FT | ~~FF~~ |

(2) A or B

| | T | F |
|---|---|---|
| 1 | TTT | FTF |
| 2 | ~~TTT~~ | TFT |
| 3 | ~~TTT~~ | TTF |

(3) A and (B and C)

| | T | F |
|---|---|---|
| 1 | TTF | FFF |
| 2 | FTF | ~~FFF~~ |
| 3 | FFT | ~~FFF~~ |

(4) A or (B or C)

| | T | F |
|---|---|---|
| 1 | TTF | FFF |
| 2 | ~~TTF~~ | TFF |
| 3 | TFT | ~~TFF~~ |

(5) A and (B or C)

| | T | F |
|---|---|---|
| 1 | TTT | FTF |
| 2 | FTT | FFT |
| 3 | ~~FTT~~ | ~~FTF~~ |

(6) A or (B and C)

**Fig. 11** Virtual table for test case generation based on MCDC

value based on coverage. CDC finds the condition value on each condition and the decision point at least once with either true or false. We pre-compose the virtual table to find the condition value quickly in multiple condition data in Fig. 10. This virtual table uses to generate test case based on CDC.

Figure 11 also shows virtual table for test case generation based on MCDC. MCDC deals with each condition and the entire conditions. Therefore, we make the virtual table like CDC in Fig. 11. The strike-out in the virtual table is duplicate condition. This virtual table uses to generate test case based on MCDC.

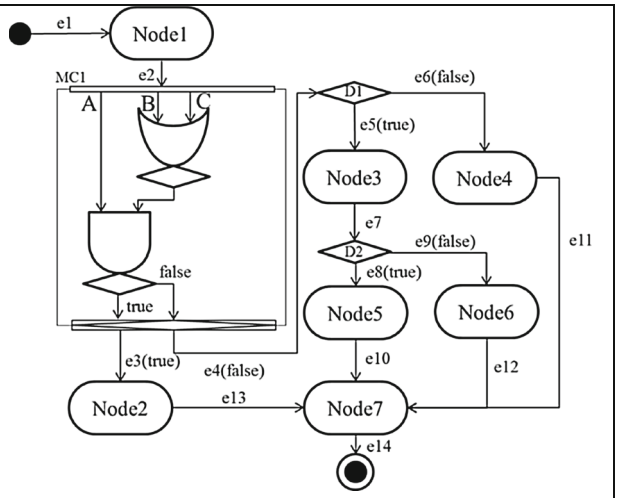
## 5 A case study

This section just shows the first and the last step of the proposed method from MCCFG to generate test cases. It shows

**Fig. 12** Example of MCCFG

```
statement Node1;
if(A and (B or C)) {
   statement Node2;
}
else {
   if(D1) {
       statement Node3;
       if(D2) {
          statement Node5;
       }
       else {
          statement Node6;
       }
   }
   else {
       statement Node4
   }
}
Statement Node7;
```

**(a)** Program Code




**(b)** MCCFG

**Table 3** The result of test case generation from Fig. 12

| Test scenario ID | Testcase ID | Inflow | Event | Condition | Outflow |
|---|---|---|---|---|---|
| (a) Statement coverage | | | | | |
| TS1 | TC1 | Node1 | e2, e3 | N/A | Node2 |
| | TC2 | Node2 | e13 | N/A | Node7 |
| TS2 | TC3 | Node3 | e7, e8 | N/A | Node5 |
| | TC4 | Node5 | e10 | N/A | Node7 |
| TS3 | TC5 | Node6 | e12 | N/A | Node7 |
| TS4 | TC6 | Node4 | e11 | N/A | Node7 |
| (b) Decision coverage | | | | | |
| TS1 | TC1 | Node1 | e2, e3 | N/A | Node2 |
| | TC2 | Node2 | e13 | N/A | Node7 |
| TS2 | TC3 | Node1 | e2, e4, e5 | N/A | Node3 |
| | TC4 | Node3 | e7,e8 | N/A | Node5 |
| | TC5 | Node5 | e10 | N/A | Node7 |
| TS3 | TC6 | Node3 | e7, e9 | N/A | Node6 |
| | TC7 | Node6 | e12 | N/A | Node7 |
| TS4 | TC8 | Node1 | e2, e4, e6 | N/A | Node4 |
| | TC9 | Node4 | e11 | N/A | Node7 |

**Table 3** continued

| Test scenario ID | Testcase ID | Inflow | Event | Condition | Outflow |
|---|---|---|---|---|---|
| (c) Condition/Decision Coverage | | | | | |
| Test Scenario ID | Testcase ID | Inflow | Event | Condition | Outflow |
| TS1 | TC1 | Node1 | e2,e3 | A == false, B == true, C==false | Node2 |
| | TC2 | Node1 | e2,e3 | A == false, B == false, C == true | Node2 |
| | TC3 | Node1 | e2,e3 | A == true, B == true, C == false | Node2 |
| | TC4 | Node1 | e2,e3 | A == true, B == false, C == false | Node2 |
| TS2 | TC5 | Node1 | e2,e4,e5 | A == false, B == true, C==false | Node3 |
| | TC6 | Node1 | e2,e4,e5 | A == false, B == false, C == true | Node3 |
| | TC7 | Node1 | e2,e4,e5 | A == true, B == true, C == false | Node3 |
| | TC8 | Node1 | e2,e4,e5 | A == true, B == false, C == false | Node3 |
| | TC9 | Node3 | e7,e8 | N/A | Node5 |
| | TC10 | Node5 | e10 | N/A | Node7 |
| TS3 | TC11 | Node3 | e7,e9 | N/A | Node6 |
| | TC12 | Node6 | e12 | N/A | Node7 |
| TS4 | TC13 | Node1 | e2,e4,e6 | A == false, B == true, C==false | Node4 |
| | TC14 | Node1 | e2,e4,e6 | A == false, B == false, C == true | Node4 |
| | TC15 | Node1 | e2,e4,e6 | A == true, B == true, C == false | Node4 |
| | TC16 | Node1 | e2,e4,e6 | A == true, B == false, C == false | Node4 |
| | TC17 | Node4 | e11 | N/A | Node7 |
| (d) Modified Condition/Decision Coverage | | | | | |
| Test Scenario ID | Testcase ID | Inflow | Event | Condition | Outflow |
| TS1 | TC1 | Node1 | e2,e3 | A == true, B == true, C==true | Node2 |
| | TC2 | Node1 | e2,e3 | A == true, B == true, C == false | Node2 |
| | TC3 | Node1 | e2,e3 | A == true, B == false, C == true | Node2 |
| | TC4 | Node1 | e2,e3 | A == true, B == false, C == false | Node2 |
| | TC5 | Node1 | e2,e3 | A == false | Node2 |
| TS2 | TC6 | Node1 | e2,e4,e5 | A == true, B == true, C==true | Node3 |
| | TC7 | Node1 | e2,e4,e5 | A == true, B == true, C == false | Node3 |
| | TC8 | Node1 | e2,e4,e5 | A == true, B == false, C == true | Node3 |
| | TC9 | Node1 | e2,e4,e5 | A == true, B == false, C == false | Node3 |
| | TC10 | Node1 | e2,e4,e5 | A == false | Node3 |
| | TC11 | Node3 | e7,e8 | N/A | Node5 |
| | TC12 | Node5 | e10 | N/A | Node7 |
| TS3 | TC13 | Node3 | e7,e9 | N/A | Node6 |
| | TC14 | Node6 | e12 | N/A | Node7 |
| TS4 | TC15 | Node1 | e2,e4,e6 | A == true, B == true, C==true | Node4 |
| | TC16 | Node1 | e2,e4,e6 | A == true, B == true, C == false | Node4 |
| | TC17 | Node1 | e2,e4,e6 | A == true, B == false, C == true | Node4 |
| | TC18 | Node1 | e2,e4,e6 | A == true, B == false, C == false | Node4 |
| | TC19 | Node1 | e2,e4,e6 | A == false | Node4 |
| | TC20 | Node4 | e11 | N/A | Node7 |

one example of MCCFG with seven nodes, fourteen edges, two decision nodes, and multiple conditions in Fig. 12. However, it can easily understands to use a white box example even if model based testing. Figure 12 shows an example of MCCFG.

We make a document template that expresses the test cases. The document template of test cases consists of Test Scenario ID, Test case ID, Inflow, Event, Condition, and Outflow. Test Scenario ID is a scenario path to connect with each test case. For example, TS1 is performed such as TC1, TC2.

TS2 is performed such as TC3, TC4 in table 2(a). Test case ID is an identification of the corresponding test case. The rest of the document describes as follows: (1) Inflow is a start point of flow, (2) Event is message(s), (3) Condition that the event is performed, and (4) Outflow is an end point of flow. Table 3 shows the test cases generated with MCCFG of Fig. 12.

## 6 Conclusions

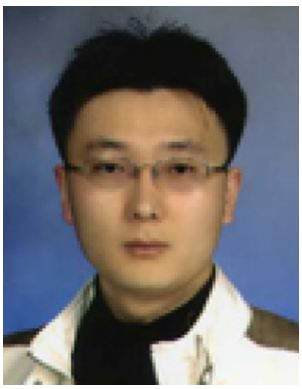
The existing CFG cannot express information about multiple conditions as white box testing technique. Therefore we propose MCCFG to extend the existing CFG for dealing with multiple conditions on model based testing, which can describe single condition and multiple condition(s) with "AND" or "OR" notation. To do this, we define graphical notation of MCCFG and adopt model transformation for generating test case from MCCFG. In addition, we design metamodel. With the proposed method, we can automatically generate test case from MCCFG for all coverage such as statement, condition, decision, condition/decision Coverage modified condition/decision, and multiple condition coverage.

Further research is extending this work with mapping between MCCFG and UML in order to apply model transformation, and method to optimize generated test case, which is not dealt in this study.

**Acknowledgements** This research was supported by the Human Resource Training Program for Regional Innovation and Creativity through the Ministry of Education and National Research Foundation of Korea (NRF-2015H1C1A1035548).

## References

1. Kim, H.N., Park, S.M., Kim, D.H.: Current technology trends in embedded software. Commun. Korean Inst. Inform. Sci. Eng. **24**(8), 5–11 (2006)
2. Jung, H.J.: The analysis of data on the basis of software test data. J. Digit. Converg. **13**(10), 1–7 (2015)
3. Kwon, W.I.: The necessity of testing software for software quality improvement. FKII Digit. **365**, 66–69 (2008)
4. Burnstein, I.: Parctical Software Testing. Springer, New York (2003)
5. Beizer, B.I.: Software Testing Techniques. Dreamtech Press, New Delhi (2002)
6. Kwon, W.I., Park, C.E.Y., Lee, H.J., Cho, H.I.: Practical Software Testing Foundation. Software Testing Alliances (2008)
7. Amland, S.: Risk-based testing: risk analysis fundamentals and metrics for software testing including a financial application case study. J. Syst. Softw. **53**, 287–295 (2000)
8. Pezze, M., Young, M.: Software Testing and Analysis: Process, Principles, and Techniques. Wiley, New York (2008)
9. OMG, MOF 2.0/XMI Mapping, v2.1.1, OMG Available Specification (2007)
10. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: eclipse modeling framework. Addison-Wesley, Reading (2008)
11. Muchnick, S.: Advanced Compiler Design and Implementation, 1st edn. Morgan Kaufmann, San Francisco (1997)
12. Chilenski, J.J., Miller, S.P.: Applicability of modified condition/decision coverage to software testing. Softw. Eng. J. **9**, 193–200 (1994)
13. Yang, Q., Li, J.J., Weiss, D.M.: A survey of coverage-based testing tools. Comput. J. **52**(5), 589–597 (2009)
14. Rountev, A., Kagan, S., Sawin, J.: Coverage criteria for testing of object interactions in sequence diagrams. In: Proceedings of Conference on Fundamental Approaches to Software Engineering, pp. 289–304 (2005)
15. Cardoso, J., Sibertin-Blanc, C.: Ordering actions in sequence diagrams of UML. In: Proceedings of International Conference on Information Technology Interfaces, pp. 3–14 (2001)
16. Garousi, V., Briand, L.C., Labiche, Y.: Control flow analysis of UML 2.0 sequence diagrams. ECMDA-FA 2005. LNCS, vol. 3748, pp. 160–174 (2005)
17. OMG, Unified Modeling Language Superstructure Version 2.4, ptc/2010-11-14



**Hyun Seung Son** received the B.S., M.S., and Ph.D degree in Software Engineering from Hongik University, Korea in 2015. His research interests are in the areas of Automation Tool Development in Embedded Software, Real Time Operation System Development, Metamodel design, and Model Transformation, Model Verification & Validation Method.

**Young B. Park** received the M.S. and Ph.D. degree from the department of Computer Science, N. Y. Polytechnic (NYU-Poly) in 1991. He is currently a professor in Dankook University. His research interests are in the areas of Intelligent Software Engineering, Automatic Software Testing, Software Development Process Enhancement and Software Refactoring.

**R. Young Chul Kim** received the B.S. degree in Computer Science from Hongik University, Korea in 1985, and the Ph.D. degree in Software Engineering from the department of Computer Science, Illinois Institute of Technology (IIT), USA in 2000. He is currently a professor in Hongik University. His research interests are in the areas of Test Process, Model Based Testing, Metamodel, and Software Process (SP).