Nov. 2006, Vol.21, No.6, pp.965–972 J. Comput. Sci. & Technol.

Code Based Analysis for Object-Oriented Systems

Swapan Bhattacharya[1] and Ananya Kanjilal[2]

[1] *Department of Computer Science and Engineering, Jadavpur University, Kolkata-700032, India*

[2] *Department of Information Technology, B. P. Poddar Institute of Management and Technology, Kolkata-700052, India*

E-mail: bswapan2000@yahoo.co.in; ag_k@rediffmail.com

Received July 15, 2005; revised May 29, 2006.

Abstract The basic features of object-oriented software makes it difficult to apply traditional testing methods in object-oriented systems. Control Flow Graph (CFG) is a well-known model used for identification of independent paths in procedural software. This paper highlights the problem of constructing CFG in object-oriented systems and proposes a new model named Extended Control Flow Graph (ECFG) for code based analysis of Object-Oriented (OO) software. ECFG is a layered CFG where nodes refer to methods rather than statements. A new metrics — Extended Cyclomatic Complexity (E-CC) is developed which is analogous to McCabe's Cyclomatic Complexity (CC) and refers to the number of independent execution paths within the OO software. The different ways in which CFG's of individual methods are connected in an ECFG are presented and formulas for E-CC for these different cases are proposed. Finally we have considered an example in Java and based on its ECFG, applied these cases to arrive at the E-CC of the total system as well as proposed a methodology for calculating the basis set, i.e., the set of independent paths for the OO system that will help in creation of test cases for code testing.

Keywords object-oriented testing, extended control flow graph, extended cyclomatic complexity, test paths, graph-based testing

1 Introduction

Traditional testing methods have been successfully implemented in procedural systems. One of the most commonly used examples of white-box testing technique is "basis path testing" which ensures that every path of a program is executed at least once. McCabe's cyclomatic complexity (CC) metric determines the number of linearly independent paths through a piece of software using the control flow graph (CFG) to determine a set of test cases which will cause executable statements to be executed at least once[1]. Evaluation of cyclomatic complexity is easy in traditional procedure oriented systems but application of the same in object-oriented context becomes difficult due to the basic properties of object oriented software — encapsulation, inheritance and polymorphism. Several approaches for testing object-oriented software are being proposed. Some of them are based on formal specifications based on algebraic specification, model based specification (VDM), LOTOS as mentioned in [2]. The work presented in [2–9] discuss approaches for testing and test generation and selection of tests based on formal specifications. Another set of work focus on testing based on design. Object-oriented software has a different set of metrics and works in [10, 11] discusses approaches for analyzing design complexity based on such metrics. The work in [12] discusses a novel test criterion based on UML collaboration diagrams. Even though these methods are gaining momentum for identification of errors early in the life cycle, code testing still constitutes a major part of testing effort invested during the life cycle. For object-oriented software, different code testing methods are being proposed and used as discussed in the next section.

2 Review of Related Work

Software is tested usually to achieve two goals- achieve quality by detecting and removing defects (*debug testing*) and assessing existing quality for measuring reliability (*operational testing*). The relationship between the two testing goals using a probabilistic analysis has been the focus of research work in [13]. In [14], a probe based testing technique has been designed that observes the internal details of execution. Probes are predetermined and pre-built and test coverage reports are generated at probe, method, class, inheritance, regression and dynamic binding levels. TATOO is a testing and analysis tool presented in [15] for object-oriented software and provides a systematic approach to testing tailored towards object behavior and particularly for class integration testing. In [16], a combination of use cases and cause effect graphing has lead to the development of a rigorous approach for acceptance testing which ensures function coverage as well. [17] discusses an evolutionary testing technique which is an approach for automation of structural test case design. It searches test data that fulfill given structural test criteria by evolutionary computation. In [18] Bi-Xin Li describes new techniques based on object-oriented program slicing techniques that compute the amount and width of information flow, correlation coefficient and coupling among basic components. In [19] dynamic data flow analysis in Java programs has been presented to detect data flow anomalies.

Testing distributed OO systems is another domain of research. Jessica Chen in [20] has described a proto-

Regular Paper

This research work has been partially supported by the project entitled "Design and Development of Domain Specific Evolutionary Prototypes" sponsored by All India Council for Technical Education vide Grant No. 8022/RID/NPROJ/RPS-110 of 2003-04.

type of an automated test control toolkit for a system consisting of processes communicating through CORBA and has identified the problems of new deadlocks that might be introduced depending upon the thread model used and test constraint specified. K. Saleh *et al.* have discussed the issue of synchronization anomalies in a concurrent Java program and proposed to detect those using probes in [21]. TTCN-3 is a standardized language for the specification and implementation of test cases. Two approaches for the graphical presentation and development of test cases has been discussed in [22] — GFT (graphical presentation format for TTCN-3) and UML 2.0 Testing profile.

Bertolino *et al.* presents a generalized algorithm in [23] that generates a set of paths that covers every arc in the program flow graph for branch testing of program code. [24] presents techniques to construct representations for programs with explicit exception occurrences. [25] proposes the design of a test data generation suite named ADTEST suitable for testing programs developed in Ada83. Reachability graph is one common approach for selecting test sequences for concurrent programs from labeled transitions systems (LTS). However that introduces a state explosion problem which is overcome by analyzing a new model ALTS (annotated LTS). [26] presents practical coverage criterion from ALTS. Use of genetic algorithms for automatic generation of test data is discussed in [27].

3 Scope of Work

In this paper we identify the problems of application of traditional testing techniques to OO systems. We have enhanced the concept of CFG and proposed a new model — ECFG and a new set of metrics-extended cyclomatic complexity (E-CC). The features of ECFG and the different cases of connectivity of methods in an ECFG have been discussed. We also proposed a method for identification of independent paths in OO software such that each of these paths would form the basis of a test case. An example in Java is considered as a case study and ECFG is constructed and E-CC derived from it substantiates our approach.

4 Problem

Consider the piece of code in Java:

Class rectangle
{int i, j;
void area ( )
{**if** $(i > 0)$
System.out.println ("Area is" $+ i * j$);}
void perimeter ( )
{**if** $(i > 0)$
System.out.println("Perimeter is" $+ 2 * (i + j)$); }}
Class square extends rectangle
{square (int i)
$\{j = i;\}$
void perimeter ( )
{System.out.println("Perimeter is" $+ 4 * i$));
}}
Class Area
{public static void main (String args[ ])
{square s = new square(5);
s.area( ); }}

Problems encountered while constructing a CFG and deriving basis set for the class *Area* in the above program:

1) The declaration of an object (in this case s) is not simply a sequence statement as a reference is made to the constructor, which is a method (here *square*). So CFG should ideally cover the logic resident within the method. It no longer remains a linear statement but refers to another CFG.

2) The reference to method (in this case *area*) may not be easily located due to inheritance. For example, in this case the statement s.area() refers to the area method not presented in class square but in class rectangle.

3) While object creation, different constructors may be used depending on the parameter list and their order of occurrence. Hence, expansion by simple substitution is not possible as there are different methods or functions with the same names.

4) Due to polymorphism, method overriding is done in many cases (*perimeter* is overridden in square), which increases complexity even more. Here the *perimeter* method of square would be executed instead of rectangle. Drawing a CFG thus becomes difficult in the context of OO programs.

5 Extended Control Flow Graph (ECFG)

We propose a new model called Extended Control Flow Graph (ECFG) analogous to CFG for an OO system. ECFG is a layered graphical model representing a collection of CFGs of the individual methods of the OO software. The methods form the nodes of the graph and the edges are drawn if a method calls another method. Each method in itself is similar to a procedural program and has its own CFG depicting the flow of control between statements of a method. Thus every node may further be referring to another graph. Essentially ECFG has two layers — the topmost layer represents the methods of individual classes and the next layer represents the CFG's of these methods. Following are the features of ECFG:

i) The graph is similar to CFG consisting of nodes and edges between nodes except at the top level where some nodes may be disconnected. It is a series of graphs arranged in layers.

ii) Nodes in CFG refer to a statement(s) whereas in ECFG nodes refer to methods.

iii) Every method has associated graphs (CFG) and cyclomatic complexity (CC) values. Methods, not found in the required class may be a part of its parent class.

iv) Object declaration is similar to variable declaration of procedural languages but is not a sequence statement (as in CFG) since it refers to constructor method.

v) Edges between nodes are formed whenever any method calls another method. There may be different ways in which nodes are connected.

The nature of the ECFG e.g., the out-degree and in-degree of nodes, the depth depends upon the manner in which the nodes i.e., the methods are being called or referred.

6 Case Study

To explain the model we have considered a fairly large example coded in Java as a representative of an OO system. It is an applet consisting of four classes and thirty nine methods. Some of them are library methods while some are subclasses overriding the default methods. Table 1 gives a list of all methods, each assigned a unique number. They have associated CFG and their corresponding cyclomatic complexity is also shown.

Table 1. Class-Method List

Class	Method	Node No.	$V(G)$
DrawBird	DrawBird (Constructor)	1	1
	set X	2	1
	set Y	3	1
	set D	4	1
	get X	5	1
	get Y	6	1
	get D	7	1
	getBird	8	1
DrawBullet	DrawBullet (Constructor)	9	1
	set X	10	1
	set Y	11	1
	set D	12	1
	get X	13	1
	get Y	14	1
	get D	15	1
	setGunman	16	1
	setBird	17	1
	setBullet	18	1
	setMsg	19	1
	setScore	20	1
Game Area	setBulletCount	21	1
	setTotalBirds	22	1
	setBirdsInScreen	23	1
	setExitFlag	24	1
	setStartFlag	25	1
	setDragFlag	26	1
	setReleaseFlag	27	1
	Update	28	1
	Paint	29	10
Shoot TheBird	Init	30	2
	Start	31	1
	Stop	32	1
	mouseDragged	33	2
	mouseReleased	34	2
	mouseClicked	35	2
	actionPerformed	36	5
	gameStart	37	2
	gameOver	38	2
	Run	39	15

Consider the source code for ShootTheBird class which consists of variable and object declarations in the beginning, followed by ten methods (30–39). Since in object-oriented programs there is no sequential flow between methods, i.e., in this case any method can execute and in any sequence depending upon the occurrence of events. Hence topmost level consists of a group of ten disconnected nodes.

Level 1: A group of ten disconnected nodes (30–39). Each node is actually a method, which have associated graphs e.g., node 36—*ActionPerformed* is an overridden library method, and hence needs to be tested.

Level 2: Some nodes have respective graphs, graph for node is shown in Fig.1.

```
36 ActionPerformed.
36.1 public void actionPerformed(ActionEvent e)
     {if (e.getSource() == newGame)
37 {gameStart();}
36.2 if (e.getSource() == pauseGame)
36.3 {pauseFlag = true;
       NewSound.stop();
       Bird.suspend();
       Bullet.suspend(); }
36.4 if (e.getSource() == resumeGame)
36.5 {pauseFlag = false;
       NewSound.play();
       Bird.resume();
       Bullet.resume(); }
36.6 if (e.getSource() == exitGame)
       {newSound.stop();
       ExitSound.play();
19   Area.setMsg("GAME OVER!");
38   GameOver();
36.7 }}
```

The CC for the graph is shown in Fig.1 is 5.

Level 3: Nodes 37 and 38 are referred to in the graph of node 36 in Fig.1. These may be further blown up into their individual CFG. For brevity this cannot be shown. There is reference to other methods of the same class as well as other classes. The entire software is a collection of CFG's linked with each other hierarchically as shown in Fig.6.

7 Implementation of ECFG

As discussed earlier, ECFG is a directed graph consisting of nodes representing methods of an OO system and the directed line denotes the manner in which these methods are linked up. One of the important and commonly used matrices for representation and study of digraphs is adjacency matrix[28]. We have used this matrix for the representation of ECFG. A simple OO system consisting of ten methods is considered. The corresponding ECFG is shown in Fig.2. The corresponding adjacency matrix is as shown in Fig.3.

The entries in the cells are either 0 or 1. Any cell may be identified as $A[i, j]$ where i denotes the node from which an edge is directed outwards and j denotes the node where an edge is incident.

Therefore $A[i, j] = 1$ or 0 if edge from i-th node to j-th node is present or absent respectively. The summation of row entries in any particular row denotes the out-degree of that node. And similarly the sum along columns denotes the in-degree of a node.

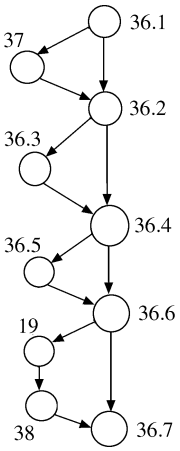

Fig.1

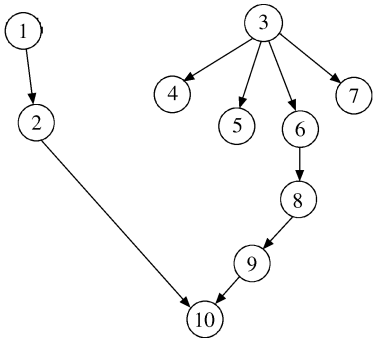

Fig.2

Nodes	1	2	3	4	5	6	7	8	9	10
1		1								
2										1
3				1	1	1	1			
4										
5										
6								1		
7										
8									1	
9										1
10										

Fig.3

8 Extended Cyclomatic Complexity

In an ECFG, the methods (or CFGs) may be connected in one of the six possible ways.

Case 1. Methods called one after another or in sequence [5, 8, 9].

Case 2. Method embedding multiple methods. [3 embedding 4, 5, 6, 7].

Case 3. Method embedding a method more than once.

Case 4. Method being embedded by different methods [10 embedded by 2 and 9].

Case 5. Methods in recursion.

Case 6. Multiple methods in recursion.

The cases appearing in the example of Fig.4 are shown within brackets alongside.

E-CC, the composite complexity of the two or more graphs taken together is distinct in each case. The physical significance of CC i.e., the number of independent paths is maintained while calculating E-CC as well.

Case 1. When two or more graphs are connected in series, i.e., the methods execute in sequence one after the other. If $V(G1)$, $V(G2), \ldots, V(Gn)$ are complexities of each individual graphs then the composite complexity is the largest among them all.

i.e., $V(G) = V(Gx)$

if $V(Gx) > V(G1), V(G2) \ldots V(Gn)$ and $1 < x < n$.

Proof. When the graphs are in series, the paths required to cover the highest complexity graph would automatically include the paths for other graphs having lesser number of paths. □

Case 2. When two or more graphs are embedded within a graph, i.e., a method calls another method which in turn calls another and so on. If $V(G1), V(G2), \ldots, V(Gn)$ are complexities of each individual graphs and they are embedded within each other i.e., $G1$ embeds $G2$, $G2$ embeds $G3$ and so on, then the *Composite complexity* $V(G)$ is given as follows:

$$V(G) = V(G1) + V(G2) + V(G3) + \cdots + V(Gn) - n.$$

The same holds true even if there is no nested embedding, i.e., $G1$ embeds $G2, G3, \ldots, Gn$.

Proof. When the graphs are embedded, one of the paths is replaced by a graph. So resulting complexity i.e., independent paths is the sum total of complexities of all minus n (No. of paths that are replaced). □

Case 3. If a graph embeds (calls) another graph more than once, say $G1$ embeds $G2$ thrice, then $G2$ would be considered to be embedded only once since once tested it needs not be tested again.

Composite complexity $V(G)$ will be the same as in Case 2 where all the complexity values refer to unique graphs— $G1, G2, \ldots, Gn$.

Case 4. When many graphs embed (call) the same graph i.e., more than one method call the same method. In this case too, the repeated graph should be considered only once. However, the point of entry should be tested in every context. All graphs calling the same graph will have one of their paths replaced by the method call, so one path should be excluded from each.

Composite complexity $V(G)$ is

$$\begin{aligned} V(G) &= (V(G1) + V(G2) - 1) + (V(G3) - 1) + \cdots \\ &\quad + (V(Gn) - 1) + (n - 2) \\ &= V(G1) + V(G2) + V(G3) + \cdots + V(Gn) - 1 \end{aligned}$$

where $V(G1)$ is the complexity for graph that is embedded multiple times and $V(G2) \cdots V(Gn)$ are complexity values for graphs which embed $G1$ and $n-2$ is the number of graphs embedding $G1$ except one.

Proof. The embedded graph $G1$ needs to be tested only once and hence it is considered only once and composite complexity is calculated. However for all the other graphs embedding $G1$ one path depicting entry point should be tested, but one path is excluded which represents the point of method call. □

Case 5. When one graph is recursively repeated i.e., a method is recursively called. *Composite complexity* $V(G) = V(G1) + 1$ where $V(G1)$ is the complexity of the method being recursively used.

Proof. Recursion represents a decision statement and hence represents two paths — either method is recursively called again or it exits from recursive loop. The entry path is replaced by the graph of the method itself and hence one path corresponding to exit needs to be tested (Hence plus one). □

Case 6. When recursion involves more than one method. This is a combination of 1, 2, 3.

Proof. Composite complexity would be the complexity of the methods taken together (following Case 1 or Case 2) and plus two for recursion as in Case 3. □

9 Identification of Different Cases of Method Connectivity

It is evident that out of the possible six cases Case 3 is not unique and need not be considered for identification of test vectors. Similarly Case 6 may be derived as a combination of other cases.

Hence we would consider only four of the following unique cases:

Case 1. For methods in sequence. Any standard path detection algorithms may be used to determine a series of nodes connected in sequence.

Case 2. Method embedding multiple methods means a node with out-degree > 1. This case is easily identifiable from adjacency matrix where for any row i, the $\Sigma i > 1$ holds true. In our example Node 3 along with 4, 5, 6, 7 is a case under Case 2.

Case 3. Method being embedded by different methods means a node having in-degree > 1. This can be identified from adjacency matrix where for any column j, the $\Sigma j > 1$ holds true. Node 10 is a case under Case 3.

Case 4. Recursion can be of two types:

- A method calling itself. This case can be detected for those nodes where the diagonal element is 1, i.e., if for any $i = j$, $A[i, j] = 1$, then it means that the i-th node is in self recursion.
- A series of methods in recursion. This case can be detected using any cycle-detection algorithm from the matrix.

10 Derivation of Basis-Set

Once the cases have been identified from the matrix, the composite complexity can be calculated and the number of test cases for that section is known. That gives the number of test cases in the basis set. The basis set calculation is described in this section.

10.1 Basis Set for Methods in Sequence

Let us assume that the basis set is a set of paths $\boldsymbol{P}$ such that $\boldsymbol{P}_i x$ denotes the paths of method i, x varying from 1 to n where n is the cyclomatic complexity of the method i.e., the number of independent paths.

If M_1 and M_2 are two methods in series and cyclomatic complexities are X and Y respectively, then the basis set are:

$$M_1 : \boldsymbol{P}_1 1, \boldsymbol{P}_1 2, \boldsymbol{P}_1 3, \boldsymbol{P}_1 4, \cdots, \boldsymbol{P}_1 x.$$
$$M_2 : \boldsymbol{P}_2 1, \boldsymbol{P}_2 2, \boldsymbol{P}_2 3, \boldsymbol{P}_2 4, \cdots, \boldsymbol{P}_2 y.$$

The composite complexity is the maximum of X and Y. The basis set would be that much number of paths. The basis set is arrived at by forming pairs between each corresponding path of the methods till all paths are exhausted. In this case, the basis set is:

Basis set $= \{\boldsymbol{P}_1 1 \| \boldsymbol{P}_2 1, \boldsymbol{P}_1 2 \| \boldsymbol{P}_2 2, \cdots\}$

Each element denotes concatenation or join of paths from individual methods. The last element is $\boldsymbol{P}_1 x \| \boldsymbol{P}_2 y$.

In general, if N methods are connected in sequence in the order $M_1, M_2, M_3 \cdots M_N$, a two-dimensional matrix $\boldsymbol{M}$ may be formed with the paths of each method forming separate rows. The size of the matrix would be $N \times \boldsymbol{P}$ where N is the number of methods in sequence and $\boldsymbol{P}$ is the maximum number of paths of a method having highest cyclomatic complexity.

Basis set of M_1:

$$\boldsymbol{P}_1 1, \boldsymbol{P}_1 2, \boldsymbol{P}_1 3, \boldsymbol{P}_1 4 \cdots \boldsymbol{P}_1 x$$

Basis set of M_2

$$\boldsymbol{P}_2 1 \ \boldsymbol{P}_2 2 \ \boldsymbol{P}_2 3 \ \boldsymbol{P}_2 4 \ \boldsymbol{P}_2 y \ \boldsymbol{P}_2 y \cdots \boldsymbol{P}_2 y$$

Basis set of M_n

$$\boldsymbol{M} = \boldsymbol{P}_n 1 \ \boldsymbol{P}_n 2 \ \boldsymbol{P}_n 3 \ \boldsymbol{P} 4 \cdots \boldsymbol{P}_n z \ \boldsymbol{P}_n z.$$

Here in this case, $\boldsymbol{M}$ represents the set of basis set of all the methods $M_1, M_2, M_3 \cdots M_n$ along the rows and the number of columns $\boldsymbol{P} = \max(x, y, \ldots z)$. In this case $\boldsymbol{P} = x$. For all rows where the number of elements is less than $\boldsymbol{P}$, the elements would be made equal to the last element in that row as shown in the above matrix $\boldsymbol{M}$.

The test vector set as a function of the basis set of individual methods is concatenation/join of elements along columns.

$$\begin{aligned}\boldsymbol{V} = \{&\boldsymbol{P}_1 1 - \boldsymbol{P}_2 1 - \boldsymbol{P}_3 1 - \cdots - \boldsymbol{P}_n 1,\\ &\boldsymbol{P}_1 2 - \boldsymbol{P}_2 2 - \boldsymbol{P}_3 2 - \cdots - \boldsymbol{P}_n 2,\\ &\boldsymbol{P}_1 3 - \boldsymbol{P}_2 3 - \boldsymbol{P}_3 3 - \cdots - \boldsymbol{P}_n 3,\\ &\vdots\\ &\boldsymbol{P}_1 x - \boldsymbol{P}_2 y - \cdots - \boldsymbol{P}_n z\}.\end{aligned}$$

10.2 Basis Set for Method Embedding Other Methods

In general let us consider N methods $M_1, M_2, \ldots, M_n$ are embedded and cyclomatic complexities are $x, y, \ldots, z$ respectively as in Fig.4.

The basis set is as given below:

$$M_1 : \boldsymbol{P}_1 1, \boldsymbol{P}_1 2, \boldsymbol{P}_1 3, \boldsymbol{P}_1 4, \cdots, \boldsymbol{P}_1 x.$$
$$M_2 : \boldsymbol{P}_2 1, \boldsymbol{P}_2 2, \boldsymbol{P}_2 3, \boldsymbol{P}_2 4, \cdots, \boldsymbol{P}_2 y.$$
$$M_n : \boldsymbol{P}_n 1, \boldsymbol{P}_n 2, \boldsymbol{P}_n 3, \boldsymbol{P}_n 4, \cdots, \boldsymbol{P}_n z.$$

The composite complexity is:

$(x + y - 1)$ + complexities of other methods from 1 to n + number of replaced paths $= (x + y - 1) + \cdots + z + (n - 2)$

The basis set is

$$\{(\boldsymbol{P}_1 1, \boldsymbol{P}_1 2, \cdots, \boldsymbol{P}_1 x) + (\boldsymbol{P}_2 1, \boldsymbol{P}_2 2, \ldots, \boldsymbol{P}_2 y) - \boldsymbol{P}_{1-2}\} + \cdots + (\boldsymbol{P}_n 1, \boldsymbol{P}_n 2, \ldots, \boldsymbol{P}_n z) + \boldsymbol{P}_{1-3} + \cdots + \boldsymbol{P}_{1-n}$$

where $\boldsymbol{P}_{1-2}$ denotes the test vector path within method 1 where method 2 is called/embedded and $\boldsymbol{P}_{1-3}, \cdots, \boldsymbol{P}_{1-n}$ denotes all the test vector paths within method 1 for the entry points of the methods 3 to n.

10.3 Basis Set for Method Embedded Multiple Times

In general let us consider N methods $M_1, M_2, \cdots, M_n$ are embedding $\boldsymbol{M}$ and the cyclomatic complexities are $x, y, \ldots, z$ and m respectively as in Fig.5.

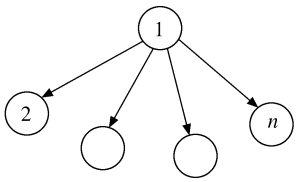

Fig.4

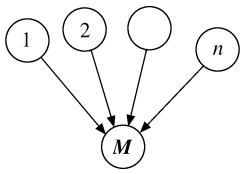

Fig.5

The basis set is as given below:

$$M_1 : \boldsymbol{P}_1 1, \boldsymbol{P}_1 2, \boldsymbol{P}_1 3, \boldsymbol{P}_1 4, \cdots, \boldsymbol{P}_1 x.$$
$$M_2 : \boldsymbol{P}_2 1, \boldsymbol{P}_2 2, \boldsymbol{P}_2 3, \boldsymbol{P}_2 4, \cdots, \boldsymbol{P}_2 y.$$
$$M_n : \boldsymbol{P}_n 1, \boldsymbol{P}_n 2, \boldsymbol{P}_n 3, \boldsymbol{P}_n 4, \cdots, \boldsymbol{P}_n z.$$
$$\boldsymbol{M} : \boldsymbol{P}1, \boldsymbol{P}2, \boldsymbol{P}3, \boldsymbol{P}4, \cdots, \boldsymbol{P}m.$$

The composite complexity is:

$$\begin{aligned}&(x + m - 1) + (y - 1) + \cdots + (z - 1) + \text{entry points}\\ &= (x + m - 1) + y + \cdots + z - (n - 1) + (n - 1)\\ &= (x + m + y + \cdots + z) - 1.\end{aligned}$$

The basis set is

$$\{(\boldsymbol{P}_1 1, \boldsymbol{P}_1 2, \ldots, \boldsymbol{P}_1 x) + (\boldsymbol{P}1, \boldsymbol{P}2, \ldots, \boldsymbol{P}m) - \boldsymbol{P}_{1-M}\}$$
$$+ (\boldsymbol{P}_n 1, \boldsymbol{P}_n 2, \ldots, \boldsymbol{P}_n z - \boldsymbol{P}_{n-M}) + \boldsymbol{P}_{2-M} + \cdots + \boldsymbol{P}_{n-M}$$

where $\boldsymbol{P}_{1-M}$ denotes the test vector path within method 1 where $\boldsymbol{M}$ is called/embedded and $\boldsymbol{P}_{2-M} \cdots \boldsymbol{P}_{n-M}$ denotes all the test vector paths within methods $2 \cdots n$ showing the entry points of method $\boldsymbol{M}$.

10.4 Basis Set for Methods in Recursion

Let us consider method M_1 in self recursion. Assuming the cyclomatic complexity of M_1 is x then the basis set is

$$M_1 : \boldsymbol{P}_1 1, \boldsymbol{P}_1 2, \boldsymbol{P}_1 3, \boldsymbol{P}_1 4, \cdots, \boldsymbol{P}_1 x.$$

The composite complexity of the method in recursion $= x + 1$ (Recursion refers to a decision statement which has entry and exit paths. The graph is embedded in the entry path, hence 1 new path corresponding to exit path needs to be tested apart from the graphs paths).

The basis set of M_r (method in recursion) is

$$M_r : (\boldsymbol{P}_{\text{entry}} - \boldsymbol{P}_1 1, \boldsymbol{P}_1 2, \boldsymbol{P}_1 3, \boldsymbol{P}_1 4, \cdots, \boldsymbol{P}_1 x - \boldsymbol{P}_{\text{entry}}) + \boldsymbol{P}_{\text{exit}}$$

where $\boldsymbol{P}_{\text{entry}}$ and $\boldsymbol{P}_{\text{exit}}$ denote the two paths entering the recursive loop and outside that respectively.

11 Results and Analysis for Case Study

We have considered a fairly large example in Java for our case study. The example is a Java applet "Shoot-TheBird" which consists of ten birds flying in the applet space, four at a time.

A gunshooter shoots down the birds. The gunshooter can be moved by mouse and a bullet is released on mouse click. The game is won if all the birds are hit or lost if the allotted 20 bullets are over. The software consists of four classes and thirty nine methods in total. The ECFG is shown in Fig.6.

Referred to Fig.6, certain regions in the graph have been identified which is simplified further and composite complexity of those branches has been calculated.

The regions are $A, B, C, D, E, F, G, H, I$ and J. The methods within each region are:

A: method 29 embedding 5, 6, 7, 8, 3, 14, 15;

B: methods 23,22, 21, 20, 17, 18, 16, 28 are in sequence;

C: method 39 embedding 3, 13, 14, 27, 5, (19 and F in sequence), 7, 6, 10, B;

D: method 37 embedding 3, 24, 25, 27, 28, 31;

E: method 33 embedding 11, 26, 27, 28;

F: method 38 embedding 3, 24, 27, 28, 32, 39;

G: method 30 embedding 1, 9;

H: method 36 embedding D and 19 & F in sequence).

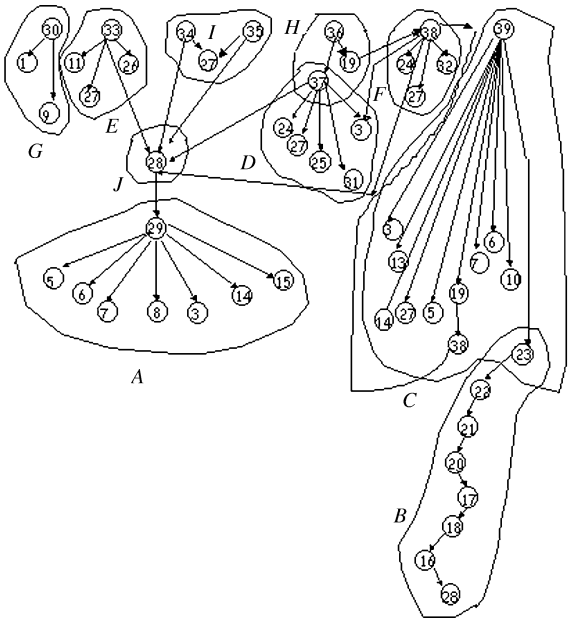

Fig.6

It is obvious from above that certain methods are embedded multiple times by many methods (called by many), e.g., 3, 19, 27, 28. Hence we consider few more regions whose composite complexity would be calculated as per Case 4. For the above regions these four methods would be excluded.

I: method 27 is embedded by 33, 34, 35, 37, 38, 39;

J: method (28 embeds A) embedded by 33, 34, 35, 37, 38, 39;

K: method 3 embedded by 29, 37, 38, 39;

L: method 19 embedded by 36, 39.

Moreover F is embedded once in C and in H. The composite complexity calculation of C and H should accordingly consider F only once (as per Case 4). The composite complexity for each of the regions is calculated as follows, the individual complexity of each method is

given as $V(G)$ in Table 1. For regions A to H, the methods 3, 19, 27, 28 are removed as they are considered separately under I, J, K, L.

A: As per Case 2, E-CC is $V(G) = V(G1) + V(G2) + \cdots - (n-1)$ (No. of graphs embedded i.e., No. of paths replaced)

$$V(A) = V(29) + V(5) + V(6) + V(7) + V(8) + V(14) + V(15) - 6 = 10 + 6 - 6 = 10.$$

B: As per Case 1, composite complexity is the maximum among them.

$$V(B) = \max[V(23), V(22), V(21), V(20), V(17), V(18), V(16)] = 1.$$

C: As per Case 2, composite complexity of 19 and F is $\max[V(19), V(F)]$

$$V(C) = V(39) + V(13) + V(14) + V(5) + V(7) + V(6) + \max[V(19), V(F)] + V(10) + V(B) - 8 = 15 + 1 + 1 + 1 + 2 + 1 + 1 + 1 + 1 - 8 = 16.$$

D: method 37 embedding 24, 25, 31.

As per Case 2, composite complexity is

$$V(D) = V(37) + V(24) + V(25) + V(31) - 3 = 2 + 1 + 1 + 1 - 3 = 2.$$

In this manner composite complexities of other regions are calculated as given below:

$$V(E) = V(33) + V(11) + V(26) - 2 = 2 + 1 + 1 - 2 = 2,$$

$$V(F) = V(38) + V(24) + V(32) + V(39) - 3 = 2 + 1 + 1 + 1 - 3 = 2,$$

$$V(G) = V(30) + V(1) + V(9) - 2 = 2 + 1 + 1 - 2 = 2,$$

$$V(H) = V(36) + \max[V(19), V(F)] + V(D) - 2 = 5 + 2 + 2 - 2 = 7.$$

I: method 27 is embedded multiple times by 33, 34, 35, 37, 38, 39

$$V(I) = (V(33) + V(27) - 1) + V(34) + V(35) + V(37) + V(38) + V(39) = (1 + V(27) - 1) + V(34) + V(35) + 1 + 1 + 1 = 1 + 2 + 2 + 3 = 8,$$

$$V(J) = (V(33) + V(28, A) - 1) + V(34) + V(35) + V(36) + V(37) + V(38) = 10 + 5 = 15,$$

$$V(K) = (V(29) + V(3) - 1) + V(39) + V(37) + V(38),$$

$$V(K) = V(3) + 3 = 1 + 3 = 4;$$

L: method 19 embedded by 39, 36

$$V(L) = (V(39) + V(19) - 1) + V(36) = 2.$$

The Extended Control Flow Graph (E-CFG) consists of the subgraphs C, E, F, G, H, I, J, K and L after simplification. Each represent certain set of independent paths which should be considered while testing.

The overall Extended Cyclomatic Complexity (E-CC) is therefore

$$\text{E-CC} = V(C) + V(E) + V(F) + V(G) + V(H) + V(I) + V(J) + V(K) + V(L) = 16 + 2 + 2 + 2 + 7 + 8 + 15 + 4 + 2 = 58.$$

This value of E-CC suggests that there are 58 possible independent paths in the system, which should be considered while testing.

This suggests that there are 58 possible independent paths in the system, which should be considered while testing. For simplicity we have considered only region G where method 30 embeds methods 1 and 9. We will be identifying test paths for this region as an example. The subgraph G and its corresponding adjacency matrix is shown in Figs.7 and 8.

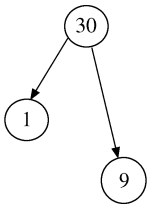

Fig.7

Nodes	1	9	30
1			
9			
30	1	1	

Fig.8

The CC and basis set of these methods are $V(30) = 2$;
Let $\boldsymbol{P}_{30-1}$ and $\boldsymbol{P}_{30-2}$ are the two test paths

$V(1) = 1$, let $\boldsymbol{P}_{1-1}$ be the path;

$V(9) = 1$, let $\boldsymbol{P}_{9-1}$ be the path.

From the adjacency matrix it is evident that out-degree of node $30 > 1$, hence it falls under Case 2. As per Case 2, E-CC is

$$V(G) = (V(30) + V(1) - 1) + V(9) - 1 = 2 + 1 - 1 + 1 - 1 = 2.$$

Methods 30, 1 and 9 are connected such that 1 and 9 are called from 30 within paths $\boldsymbol{P}_{30-1}$ and $\boldsymbol{P}_{30-2}$ respectively. Hence, test vector set:

$$\boldsymbol{V} = \{\boldsymbol{P}_{30-1} - \boldsymbol{P}_{1-1}, \boldsymbol{P}_{30-2} - \boldsymbol{P}_{9-1}\}$$

where $\boldsymbol{P}_{30-1} - \boldsymbol{P}_{1-1}$ and $\boldsymbol{P}_{30-2} - \boldsymbol{P}_{9-1}$ are the paths depicting method calls.

This is just a generalized format. More specifically the test paths $\boldsymbol{P}_{30-1}$, $\boldsymbol{P}_{1-1}$, $\boldsymbol{P}_{9-1}$ would be replaced by the basis set of the respective methods as derived from their corresponding CFGs. The cases can be applied to all the regions and likewise test vectors may be generated.

12 Conclusion

In this paper we propose a graph based methodology for analysis of OO systems focusing on the structure of program code and develop an analogous model to CFG for analysis and testing of OO systems named Extended Control Flow Graph (ECFG). ECFG is a layered model with the top layer consisting of several methods and each method or node in turn can be exploded as the second layer of graph. The different possible cases in which

the nodes or methods may be connected are discussed and a new set of metrics has been proposed named Extended Cyclomatic complexity (E-CC). It identifies the minimum number of independent paths in the object-oriented software. The methodology for implementation of ECFG and identification of methods connectivity has also been discussed. The methodology discussed may be applied towards development of graph-based tools for testing object-oriented software.

References

[1] Ghezzi C, Jazayeri M, Mandrioli D. Fundamentals of Software Engineering. India: Prentice Hall, 1998.

[2] Marie-Claude Gaudel. Testing from formal specifications, a generic approach. *LNCS 2043*, Springer-Verlag, May 2001, pp.35–48.

[3] Tse T H, Zhinong Xu. Test case generation for class-level object-oriented testing. In *Quality Process Convergence: Proc. 9th Int. Software Quality Week (QW'96)*, San Francisco, California, 1996, pp.4T4.0–4T4.12.

[4] Tai K C, Lie Y. A test generation strategy for pairwise testing. *IEEE TSE,* January 2002, 28(1): 109–111.

[5] Richard H Carver, Kuo-Chung Tai. Use of sequencing constraints for specification-based testing of concurrent programs. *IEEE TSE,* June 1998, 24(6): 471–490.

[6] Murali Rangarajan, Perry Alexander, Nael Abu-Ghazaleh. Using automatable proof obligations for component based design checking. In *Proc. IEEE Conf. Workshop on Engineering of Computer-Based Systems,* Los Alamitos, USA, March 1999, p.304.

[7] Stephane Barbey, Didier Buchs, Cecile Peraire. A theory of specification based testing for object-oriented software. In *Proc. European Dependable Computing Conference (EDCC2)*, Taormina, Italy, October 1996.

[8] Huo Yan Chen, T H Tse, Yue Tang Deng. ROCS: An object-oriented class-level testing system based on the relevant observable contexts technique. *Information and Software Technology,* July 2000, 42(10): 677–686.

[9] Betty H C Cheng, Enoch Y Wang. Formalizing and integrating the dynamic model for object-oriented modeling. *IEEE Trans. Software and Engineering,* August 2002, 28(8): 747–762.

[10] Rajendra K Bandi, Vijay K Vaishnavi, Daniel E Turk. Predicting maintenance performance using object-oriented design complexity metrics. *IEEE Trans. Software and Engineering,* January 2003, 29(1): 77–87.

[11] Ramanath Subramanyam, Krishnan M S. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Software and Engineering,* April 2003, 29(4): 297–310.

[12] Aynur Abdurazik, Jeff Offut. Using UML collaboration diagrams for static checking and test generation. *LNCS 1939/2000,* UML 2000: The Unified Modeling Language Advancing the Standard, January 2000, p.383.

[13] Phyllis G Frankl, Richard G Hamlet, Bev Littlewood, Lorenzo Strigini. Evaluating testing methods by delivered reliability. *IEEE Trans. Software and Engineering,* August 1998, 24(8): 586–601.

[14] Anita Goel, Gupta S C, Wasan S K. Probe mechanism for object-oriented software testing. In *Proc. FASE 2003, 6th Int. Conf.*, Warsaw, Poland, April 7–11, 2003, pp.310–324.

[15] Amie L Souter, Tiffany M Wong, Stacey A Shindo, Lori L Pollock. TATOO: Testing and analysis tool for object-oriented software. In *Proc. the 7th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, part of *ETAPS*, Genova, Italy, April, 2001, pp.389–403.

[16] Jose L Fernandez. Acceptance testing of object-oriented systems. In *Proc. Ada-Europe Int. Conf. Reliable Software Technologies, LNCS 1622,* 1999, pp.114–123.

[17] Joachim Wegener, Andre Baresel, Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology,* 2001, 43(14): 841–854.

[18] Bixin Li. A technique to analyze information-flow in object oriented programs. *Information and Software Technology,* Apr. 2001, 45(6): 841–854.

[19] Boujarwah A S, Saeh K, Al-Dalla J. Dynamic data flow analysis for Java programs. *Information and Software Technology,* August 2003, 42(11): 765–775.

[20] Jessica Chen. On using static analysis in distributed system testing. *LNCS Vol 1999/2001,* Springer-Verlag, ISSN-0302-9743, Jan. 2001, p.145.

[21] Saleh K, Abdel Aziz Boujarwah, Jehad Al-Dallal. Anomaly detection in concurrent Java programs using dynamic data flow analysis. *Information and Software Technology,* December 2001, 43(15): 973–981.

[22] Schieferdecker I, Jens Grabowski. The graphical format of TTCN-3 in the context of MSC and UML. *LNCS, Vol 2599/2003,* Springer-Verlag, January 2003, pp.233–252.

[23] Bertolino A, Marre M. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Trans. Software and Engineering,* 1994, 20(12): 885–899.

[24] Saurabh Sinha, Mary Jean Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Trans. Software and Engineering,* September 2000, 26(9): 849–871.

[25] Matthew J Gallagher, V Lakshmi Narasimhan. ADTEST: A test data generation suite for Ada software systems. *IEEE Trans. Software and Engineering,* Aug. 1997, 26(9): 473–484.

[26] Pramod V Koppol, Richard H Carver, Kuo-Chung Tai. Incremental integration testing of concurrent programs. *IEEE Trans. Software and Engineering,* June 2002, 28(6): 607–623.

[27] Michael C C, Mcgraw G, Schatz M A. Generating software test data by evolution. *IEEE Trans. Software and Engineering,* December 2001, 27(12): 1085–1110.

[28] Narsingh Deo. Graph Theory. India: Prentice Hall, 2003.

Swapan Bhattacharya is presently a faculty member in the Department of Computer Science & Engg., Jadavpur University, Kolkata, India. He received his Ph.D. degree in computer science in 1991 from University of Calcutta, India. His areas of research interests are distributed computing and software engineering. He received Young Scientist Award from UNESCO in 1989. As a Sr. Research Associate of National Research Council, USA, he had also served as the coordinator of Ph.D. program in Software Engg. in Naval Postgraduate School, Monterey, CA during 1999–2001. He has published about 100 research papers in various international platforms. He is also actively involved in organizing international conferences on software engineering and distributed computing. He may be reached at bswapan2000@yahoo.co.in.

Ananya Kanjilal is presently a faculty member in the Department of Information Technology, B.P. Poddar Institute of Managmement & Technology affiliated to West Bengal University of Technology, Kolkata, India. She is pursuing Ph.D. degree under the supervision of Professor Swapan Bhattacharya from Jadavpur University, India. Her areas of research interests are Object Oriented Systems and Software Engineering. Her total number of publication in various international platforms is 5. She may be reached at ag_k@rediffmail.com.