Compiler Based Test Case Generation

Vipin Kumar K S¹, Dr. Sheena Mathew²
¹ Dept of CSE, Govt. Engg. College, Thrissur, Kerala, India.
   Email: vipin.kumar.k.s@gmail.com
² Dept of CSE, School of Engg., CUSAT, Kalamassery, Kerala, India
   Email: sheenamathew@cusat.ac.in

Abstract—This paper presents a compiler based approach to automate test case generation. This approach relies on object oriented metrics to identify complex message method sequences in the program. In this approach the compiler compiles the code to generate a graphical model for the object oriented program based on System Dependence Graph. The SDG is augmented with other important information like control flow and message-method sequence information to create the model. The compiler also computes the complexity metrics for the compiled program. The message method sequence information represented in the model is used to come up with a cumulative complexity measure for each message method sequence. The compiler then outputs test scripts representing test cases for most complex message method sequences in the program. In this approach, automated analysis of the code results in increased efficiency of analysis. Different aspects of an object oriented program that contribute to complexity are evaluated to estimate complexity of a program segment.

Index Terms—OOP, test case, analysis, object oriented, metrics, compiler based, SDG, CIDG.

I. INTRODUCTION

The need for development of large systems utilizing minimum resources has accelerated the introduction of object oriented programming concepts into a wide variety of applications such as real time systems, control systems, large business system, and other enterprise systems. Ease of modeling and design of object oriented programs (OOPs) facilitate ease of development of these systems using OOP concepts. Software metrics are used to improve and estimate software quality, complexity, cost and productivity. This paper introduces a compiler based evaluation of software metrics to identify complex classes in an OOP. Although current metrics and models are certainly inadequate, a number of organizations are achieving promising results through their use. Results should improve further as we gain additional experience with various metrics models. The first definition of software metrics is proposed by Norman Fenton: [2] software metrics is a collective term used to describe the very wide range of activities concerned with measurement in software engineering. These activities range from producing numbers that characterize properties of software code (these are the classic software ‘metrics’) through to models that help predict software resource requirement and software quality. The subject also includes the quantitative aspects of quality control and assurance – and this covers activities like recording and monitoring defects during development and testing. The testing of software involves taking important managerial decisions based on risk assessment [3]. The proposed system is shown in Fig 1.
This paper is organized as follows:

- Development of the model by augmenting the basic CIDG.
- Use of this model for identifying test cases.

![Proposed system for test case generation.](image)

**III. MODEL FOR OBJECT ORIENTED PROGRAM**

Model for the program is constructed by extending SDG for object oriented programs. The model proposed in [13] is extended for identification of test cases. Class Dependence Graph (CIDG) represents the data and control dependencies between different program elements. CIDG is augmented with the control flow information, exception information and message-method sequences to create the model.

The SDG is extended with the control flow information so as to enable us to perform a better statement level analysis of the object oriented program. This is particularly helpful in identifying the sequence of statements that form execution trace of a test case. The model may be augmented with message method sequence information by analyzing the UML sequence diagrams for the program or by analyzing the code during compilation.

The control flow is represented by introducing control flow edges to specify the ordering of statements within a given method. The methods invoking other methods along with the messages used for the invocation are represented in the method sequences. The method invocation edges connect the method entry nodes of calling method to that of the called method, which form the individual edges of a method sequence. The control flow edges helps to represent how exceptions affect the normal flow of control in a program. Additional nodes are used to represent exceptions in the CIDG. The CIDG extended with this information is called as Extended CIDG or ECIDG for short. Fig 3 shows the ECIDG for the program given in Fig 2.
Fig. 2. An Example Program.

```c
int x=0;

CE1 class A{
    E2 void mA()
    S3 { int i=0;
    S4 B *bptr = new B();
    S5 cin>>a;
    S6 try {
    C7 bptr->mB(a);
    }
    S8 catch(E1 &e1){
    S9 cont<<"error E1"<<endl;
    S10 catch(...){
    S11 cont<<"x"<<endl;
    S12 cont<<"x"<<endl;
    }
    CE13 class B{
    E14 float mB(int y)
    }
    S15 try {
    S16 if (y==0)
    S17 throw new E2();
    S18 x=sqrt(y);
    } catch(E2 &e2){
    S20 cont<<"error E2"<<endl;
    S21 throw: ;
    S22 cont<<"x"<<endl;
    }
    }
    E24 main(int argc, char **argv)
    {
    S25 A *aptr = new A();
    S26 Aptr->mA();
    }
```

Fig. 3 The method sequence for the sample program in Fig 2.
III. COMPILER BASED APPROACH FOR SOFTWARE METRIC EVALUATION

Compilers are usually made with tools such as parser and lexical analysis generators. A parser-generator takes a grammar, specified in a language such as BNF or EBNF, checks it, and constructs a representation (the parser) which will execute semantic actions as phrases over the grammar are recognized.

In this paper, we deal with a compiler based approach to evaluate certain software metrics for an object oriented program. The compiler takes the program as input, parses it and along with that, evaluates the metrics for the program.

Two types of metrics can be used for software development- product metrics and process metrics. Product metrics enumerate the characteristics of the software whereas process metrics enumerate the characteristics of the process that is used to develop the software. The use of metrics can be considered as a key factor in determining the success of software engineering strategy and its management. A single metric by itself is not adequate to provide enough information on the object oriented aspects of a software. Hence, we decided to include 11 different metrics to account for the measure. They are detailed below:

A. Cyclomatic Complexity

This metric is used to find out the complexity of an algorithm. It is a count of the total number of different paths that needs to be considered to test the method completely. The metric, proposed by McCabe, is a graph-theoretic-based concept [4]. For a graph G with n nodes, e edges and p connected components, the Cyclomatic number.

\[ V(G) = e - n + p \]

Cyclomatic complexity can also be measured as —number of decision nodes + 1.

B. Depth Of Inheritance Tree(DIT)

The depth of a class within the inheritance hierarchy is the maximum length from the class node to the root of the tree, measured by the number of ancestral classes [1]. As the depth increases, it becomes more and more complex to predict the behavior of the class.

C. Lack of Cohesion of Methods (LCOM)

It gives an account of the dissimilarity between the methods of a class. There are several LCOM metrics. The LCOM HS (HS stands for Henderson-Sellers) takes its values in the range [0- 2]. A LCOM HS value highest than 1 should be considered alarming. Here are algorithms used to compute LCOM metrics:

\[ LCOM = 1 - \frac{\text{sum(MF)}}{M*F} \]

\[ LCOM \text{ HS} = \frac{(M - \text{sum(MF)}/F)(M-1)}{M} \]

Where:
M is the number of methods in class (both static and instance methods are counted, it includes also constructors, properties getters/setters, events add/remove methods). F is the number of instance fields in the class. MF is the number of methods of the class accessing a particular instance field. Sum(MF) is the sum of MF over all instance fields of the class. In our implementation, we have chosen LCOM and not LCOM HS. A high value of LCOM (i.e., closer to 1) indicates poor cohesion and hence lesser reliability. A value closer to 0 indicates good cohesion and hence greater reliability.

D. Coupling between Object Classes (CBO)

CBO for a class is defined as [1] the number of other classes to which it is coupled. Two classes are coupled when methods declared in one class use methods or instance variables of the other class. Excessive coupling is detrimental to modular design and prevents reuse. Coupling increases the complexity of the software.

E. Methods per Class

It is the average number of methods per object class. It is calculated as:

Methods per class = Total number of methods/Total number of object classes.

A large number of methods per object class complicate testing due to the increased size and complexity [5]. If the number of methods per object class gets too large, extensibility will be hard.

F. Executability Factor

This metric calculates the ratio of number of executable lines of code in the program, to the total number of lines of code[4]. Low value of Executability indicates that there are excess variable declarations or non operational statements, which perform only declarations.

G. Method Hiding Factor (MHF)

It is defined as [4]the ratio of the sum of the invisibilities of all methods defined in all classes to the total number of methods defined in the system under consideration. The invisibility of a method is the percentage of the total classes from which this method is not visible.

H. Attribute Hiding Factor (AHF)

It's defined as [4] the ratio of the sum of the inherited attributes in all classes of the system under consideration to the total number of available attributes (locally defined plus inherited) for all classes.

I. Number of Catch Blocks per class

It is defined as [9] the ratio of the catch blocks in a class to the total number of classes in the program. This helps to analyze if all the possible exceptions and their corresponding catch statements is being defined in the program.

J. Metric on methods (NbVariables)

It is the number of variables declared in the body of a method[5]. Methods where NbVariables is higher than 8, are hard to understand and maintain. Methods where NbVariables is higher than 15, are extremely complex and should be split in to smaller methods.
K. Metric on methods(NbParameters)

It is the number of parameters of a method [5]. Methods where NbParameters is higher than 5, might be painful to call and might degrade performance and reliability.

IV. VALUES FOR COMPLEXITY METRICS

The compiler identifies test critical classes based on the values of 11 different software metrics for the program. These values reflect on the complexity of each class in the program. The values also hint at conformance of each classes to good design techniques. There are some optimal ranges of value, for each software metric, which should be followed by an efficient and reliable object oriented program. By comparing against these values, valuable suggestions relating to improvement can be made.

The optimum values for the metrics are as follows as shown in Table 1:

<table>
<thead>
<tr>
<th>Name of the Metric</th>
<th>Optimal Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclomatic Complexity</td>
<td>1-10</td>
</tr>
<tr>
<td>Depth of Inheritance</td>
<td>0-4</td>
</tr>
<tr>
<td>Lack of Cohesion of Methods</td>
<td>Lesser the better</td>
</tr>
<tr>
<td>Coupling between object classes</td>
<td>0-4</td>
</tr>
<tr>
<td>Methods per class</td>
<td>Lesser the better</td>
</tr>
<tr>
<td>Executibility</td>
<td>Higher the better</td>
</tr>
<tr>
<td>Method hiding factor</td>
<td>Higher the better</td>
</tr>
<tr>
<td>Attribute hiding factor</td>
<td>Higher the better</td>
</tr>
<tr>
<td>Number of catch blocks per class</td>
<td>Higher the better</td>
</tr>
<tr>
<td>NbVariables</td>
<td>0-8</td>
</tr>
<tr>
<td>NbParameters</td>
<td>0-5</td>
</tr>
</tbody>
</table>

Optimal values of some metrics are scientifically fixed and some others were fixed for evaluation. If the values obtained for all the metrics by a program fall in the optimal ranges, we say that the program conforms to the object oriented concepts and clearly, the program can offer high performance.

V. TEST CASE GENERATION

The test case are generated in the form of test scripts that identify the statements in the message-method sequences. The test cases are identified in the form of message-method sequences from the model. The message-method sequences are identified following the message-method arcs in the ECIDG. Fig 3 shows the message-method sequence <mm1> and its corresponding arcs.

Calculation of Metric Value:

\[
\text{NORMALIZED METRIC} = \frac{(\text{METRIC} - \text{MIN METRIC})}{(\text{MAX METRIC} - \text{MIN METRIC})} \tag{1}
\]
Calculation of Complexity Score of test cases:

**FOR EACH TEST CASE**

ADD THE METRIC VALUES OF EACH OF THE INVOLVED CLASSES AND METHODS TOGETHER
ADD DU PAIR COVERAGE (0.0 - 1.0 RANGE) TO THIS TOTAL VALUE
THE FINAL VALUE IS TAKEN TO BE THE SCORE OF THE TEST CASE

\[ (2) \]

![Diagram](image)

**Fig. 4 Arbitrary method sequences.**

The Fig 4 shows three arbitrary message-method sequences which identify three different test cases. The complexity score is computed according to equations (1) and (2) which represents the priority of the test cases.

**VI. CONCLUSION**

The system identifies the various message-method sequences in the program. These message-method specification is output in the form of test scripts. These test scripts are then converted to test cases for testing the program.

**REFERENCES**


564

