Evaluation of algorithm and testing of programs using slicing methods

Sambit Kumar Mishra
Associate Professor , Department of Computer Sc.&Engg.
Ajay Binay Institute of Technology, Cuttack

Prof.(Dr.)Dulu Pattnaik
Principal, Government Engineering College, Bhawanipatna
Prof.(Dr.) Srikanta Pattnaik
Professor, S.O.A. University , Bhubaneswar

Abstract

Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it does not do anything unintended. Software should be predictable and consistent, offering no surprises to users. Software testing is easier, in some ways, because the array of software and operating systems is much more sophisticated than ever, providing intrinsic well-tested routines that can be incorporated into applications without the need for a programmer to develop them from scratch. Software testing is a technical task, but it also involves some important considerations of economics and human psychology. In an ideal world, we would want to test every possible permutation of a program. In most cases, however, this simply is not possible. Even a seemingly simple program can have hundreds or thousands of possible input and output combinations. Creating test cases for all of the possible input and output combinations are absolutely theoretical. Complete testing of a complex application would take too long and require too many human resources to be economically feasible. Testing is the process of demonstrating that errors are not present. The purpose of testing is to show that a program performs its intended functions correctly and establishes confidence that a program does what it is supposed to do.

Key words : Software testing, functional testing, white box testing, black box testing, UML, symbolic constraints, domain specification.

1. Introduction

Software process is a representative of multi-element nonlinear complicated system. It is the sequential set of procedure related to activities, constraints and resources in software life cycle. As the expansion of software development scale, traditional software resource configuration model has significant limitations. Since software development is a knowledge-intensive activity, the keynote of software engineering modeling is rational distribution of human resources. Rigorous functional testing is critical to successful application development. By automating key elements of functional testing, aggressive release schedules can be met, also can be tested more thoroughly and reliably. Functional testing provides the ability to verify that applications work as they should be expected to do. Functional tests capture user requirements in a useful way, give both users and developers confidence that the business processes meet those requirements. A test case that finds a new error can hardly be considered unsuccessful; rather, it has proven to be a valuable investment. An unsuccessful test case is one that causes a program to produce the correct result without finding any errors. In case of black-box testing, the program is viewed as a black box. In this case, goal is to be completely unconcerned about the internal behavior and structure of the program. Instead, concentrate on finding circumstances in which the program does not behave according to its specifications. The test data are derived solely from the specifications. But in case of white-box or logic-driven testing, it permits the user to examine the internal structure of the program. This strategy derives test data from an examination of the program’s logic. The goal is to establish for this strategy, the analog to exhaustive input testing in the black-box approach. Causing every statement in the program to execute at least once might appear to be the answer, but it is not difficult to show that this is highly inadequate.

2. Review of Literature

Chartchai Doungsaard et al[1] have proposed to generate test data from UML state diagram, so that test data can be generated before coding. They implemented to generate sequences of triggers for UML state diagram as test cases using genetic algorithm. His proposed algorithm has been demonstrated manually for an example of a vending machine. C. S. Krishnamoorthy et al [2] discusses the object-oriented design aside from give a more natural representation of information , he also facilitates better memory management and code reusability and his team shows how classes derived from the implemented libraries can be used for the practical size optimization of large space trusses, where several constructability aspects have been incorporated to simulate real-world design constraints. Strategies are discussed to model the chromosome and to code genetic operators to handle such constraints. Strategies are also suggested for member grouping for reducing the problem size and implementing move-limit concepts for reducing the search space adaptively in a phased manner. The implemented libraries

are tested on a number of large previously fabricated space trusses, and the results are compared with previously reported values. Federico M. Stefanini and Alessandro Camussi[3] approach becomes feasible performing a Monte Carlo simulation of the natural evolution process, in which population improvement (search for solutions) in a considered environment (the spec problem domain) is achieved by following the genetic paradigm. Starting with a randomly constituted sample of individuals, drawn from the population of admissible values and expressed as binary strings, random mating brings about individuals of the next generation. Parents are chosen with a greater probability as the number of constraints violated by each individual becomes smaller. To generate the UML state diagrams there is automatic test case, which has been suggested by Samuel et al [4]. All the steps associated with diagrams covered by this test case. Simple predicates can reduce the number of test cases. They have taken the example of an ice cream vending machine. But Samuel et al were not succeeded to achieve globally optimal solution using alternating variable method. So they proposed genetic algorithm to achieve the UML state diagrams. M. Prasanna and K.R. Chandran[5] have suggested a model based approach in dealing with object behavioral aspect of the system and deriving test cases based on the tree structure coupled with genetic algorithm.

3. Software testing principles :

1. A necessary part of a test case is a definition of the expected output or result.
2. A programmer should avoid attempting to test his or her own program.
3. A programming organization should not test its own programs.
4. Thoroughly inspect the results of each test.
5. Test cases must be written for input conditions that are invalid and unexpected, as well as for those that are valid and expected.
6. Examining a program to see if it does not do what it is supposed to do is only half the battle; the other half is seeing whether the program does what it is not supposed to do.
7. Avoid throwaway test cases unless the program is truly a throwaway program.

4. Example :

4.1. Logic driven testing :

In logic-driven testing, it permits to examine the internal structure of the program. This strategy derives test data from an examination of the program’s logic. The goal at this point is to establish, for this strategy, the analog to exhaustive input testing in the black-box approach. Causing every statement in the program to execute at least once might appear to be the answer, but it is not difficult to show that this is highly inadequate.

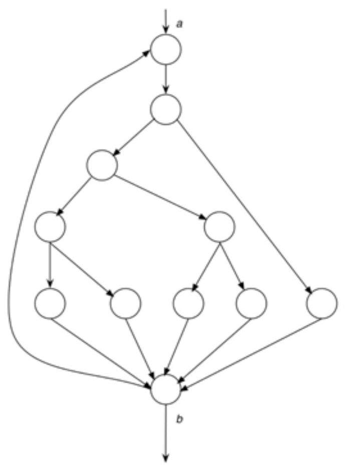

Figure-4.1.(Control flow diagram of a program)

Considering the following example,

if (a-b < c)

printf(“%d”,c); printf(%d”, a-b<c) ; it is seen that the statement contains an error because it should compare c to the absolute value of a-b. Detection of this error, however, is dependent upon the values used for a and b and would not necessarily be detected by just executing every path through the program.

4.2. Logic coverage testing :

Considering the following example, it is observed that , by setting a=2, b=0, and x=3 at point *a*, every statement may be executed once. Actually, x may be assigned any value.

```
 void  test(int a, int b, int x)
{
if (a>1 && b==0)
{
x=x/a;
}
if (a==2 || x>1) {
x=x+1;
} }
```

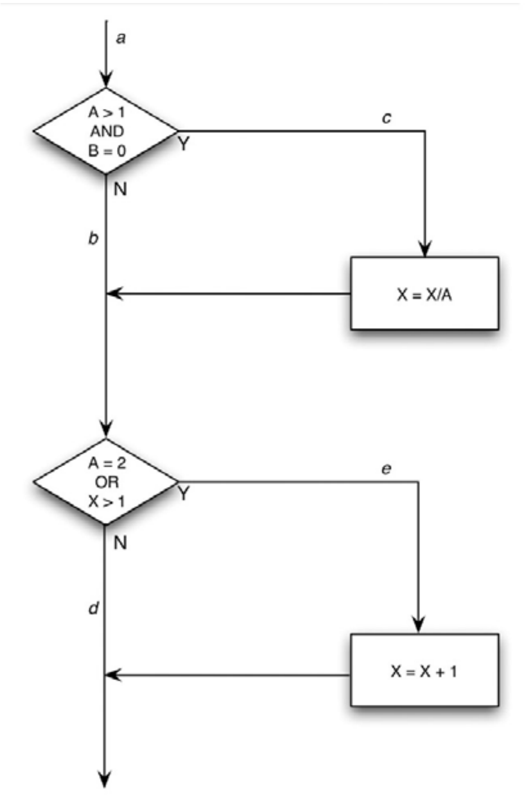

Figure-4.2. Flow chart (Logic coverage criteria)

A stronger logic-coverage criterion is known as decision coverage or branch coverage._This criterion states that you must write enough test cases that each decision has a true and a false outcome at least once. In other words, each branch direction must be traversed at least once. Examples of branch or decision statements are switch, do-while, and if-else statements. Multi way GOTO statements qualify in some programming languages. Decision coverage usually can satisfy statement coverage. Since every statement is on some sub path emanating either from a branch statement or from the entry point of the program, every statement must be executed if every branch direction is executed. However, there are at least three exceptions:

- Programs with no decisions.
- Programs or subroutines/methods with multiple entry points. A given statement might be executed only if the program is entered at a particular entry point.

- Statements within ON-units. Traversing every branch direction will not necessarily cause all accessed units to be executed.

Since the statement coverage is deemed to be a necessary condition, decision coverage must be defined to include statement coverage. Hence, decision coverage requires that each decision have a true and a false outcome, and that each statement be executed at least once. An alternative and easier way of expressing it is that each decision has a true and a false outcome, and that each point of entry be invoked at least once.

Another method called decision/condition coverage, requires sufficient test cases that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once. A weakness with decision/condition coverage is that, although it may appear to exercise all outcomes of all conditions, it frequently does not because certain conditions mask other conditions.

The results of conditions in and and or expressions can mask or block the evaluation of other conditions. For example, if an and condition is false, none of the subsequent conditions in the expression need be evaluated. Likewise if an or condition is true, none of the subsequent conditions need be evaluated. Hence, errors in logical expressions are not necessarily revealed by the condition-coverage and decision/condition-coverage criteria.

5. **Pseudocode for the analysis algorithm** :

```
1 Input: a list of available statements to be analyzed, and current symbolic constraint
2Analyze(avail_statements,symbolic_constraint):
3 for each statement
4AnalyzeStatement(avail_statements, symbolic_constraint)
5  if avail_statements=  ``v := e'' then
6  symbolic_constraint(v) := eval(Analyze avail_statement(e), symbolic_constraint)
7  else if statement = ``if(cond) then test_case-1 else test_case-2'' then
8  AnalyzeStatement(condition)
9switch_evaluation(condition, symbolic_constraint)
10 case TRUE:
11Analyze(test_case-1,symbolic_constraint)
12 case FALSE:
13Analyze(test_case-2,symbolic_constraint)
14 case SYMBOLIC:
15 if  test_case-1 is ``error block'' then
16 GenerateSideCondition(!condition)
17Analyze(test_case-2,symbolic_constraint)
18 else if test_case-2 is ``error block'' then
19 GenerateSideCondition(condition)
20Analyze(test_case-1,symbolic_constraint)
21 else
22 break
23 else if statement = ``f(a, b, ...)'' then
24 if f in code then
25Analyze(code[f(a,b,...)],symbolic_constraint)
26 else if ``f(a, b, ...)'' in domain knowledge then
27 Update(symbolic_constraint, domain knowledge[f(a, b, ...)])
28 GenerateSideCondition(``f(a,b,..)'')
29 else
30 break
```

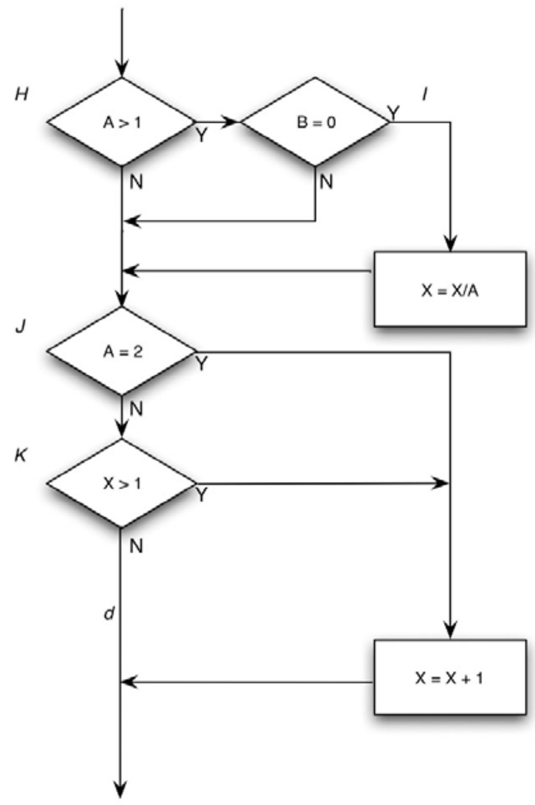

Figure 5.1. Flow chart (code analysis)

6. Experimental Analysis

By setting the parameters of a and b it is observed that the parametric value of the formal argument x varies. While simulating it is observed that the value of x is also dependent to the symbolic constraints. While evaluating the test cases 1 and 2 it is seen that the test function is valid along with the parameters a and b if the accurate domain knowledge to the symbolic constraints are provided.

Table 6.1. Parametric evaluation of arguments and symbolic constraints

Sl.No.	**Arg a**	**Arg b**	**Arg x**	**Test Function value**
1.	**2**	**1**	**1**	**0.7**
2.	**2**	**2**	**1**	**0.9**
3.	**3**	**1**	**2**	**0.47**
4.	**3**	**2**	**2**	**0.48**
5.	**3**	**2**	**3**	**0.67**

7. Conclusion and future direction

Traditional approaches to evaluate dependability of software systems in industry are process-based. In these approaches, a system is considered suitable for certification by an agency if its development adheres to one or more standards. The main criticism of process based approaches is the lack of an evident link between the extent of the quality assurance activities that are mandated by the standards and the level of dependability that is inferred. In response, a number of case-based approaches to software dependability may be adopted whose goal will be to provide an argument that will directly link the developer's claims about the dependability of the system to concrete evidence. By fast algorithms, modular methods, parallel approaches and software engineering, it is aimed at improving the theoretical and practical efficiency for solving non-linear polynomial systems symbolically by way of triangular decompositions. Functional testing need not be a time-consuming or expensive proposition. By automating functional testing, the major steps may be forwarded in the ability to improve automated processes.

8. Reference

[1] C. Doungsaard, K. Dahal, A. Hossain, and T.Suwannasart, “An Automatic Test Data Generation from UML State Diagram Using Genetic Algorithm”, Proceedings of International Conference on Software, Knowledge, Information Management and Applications (SKIMA), 2006, pp. 1-5.

[2] C. S. Krishnamoorthy, P. P. Venkatesh, and R. Sudarshan, “Object - Oriented Framework for Genetic Algorithms with Application to Space Truss Optimization”, Journal Computer in Civil Engineering, vol 16, no 1, 2002, pp. 66-75.

[3] F. M. Stefanini and A. Camussi, “APLOGEN: An Object Oriented Genetic Algorithm Performing Monte Carlo Optimization”, Oxford Journal , Life Science Bioinformatics, vol 09, no 06, 1993, pp 695- 700.

[4] P. Samuel, R. Mall, and A. K. Bothra , ”Automatic Test Case Generation Using UML State Diagrams”, IET Software, 2008, pp. 79-93.

[5] M. Prasanna1 and K. R. Chandran, “Automatic Test Case Generation for UML Object diagrams Using Genetic Algorithm”, International Journal Advance Soft Computing Application, vol 1, no 1, 2009, pp. 19-31.