Automated Testing Tool for NATURAL[®]

Sheung-lun Hung, Lam-for Kwok, Ken Chee-keung Law

Department of Computer Science, City Polytechnic of Hong Kong
83 Tat Chee Avenue, Kowloon, Hong Kong

Abstract

Traditional manual testing procedures are time-consuming and unreliable. As program gets larger and more complex, automated testing is an essential step in the software development cycle. This project aims at developing the prototype of an automated coverage testing tool for NATURAL[®] on a range of platforms including UNIX[®] and MS-Window. The prototype has proven to be applicable to a large number of programs of various sizes. This work represents a first step towards the complete automation of the coverage testing process and further development is on the way to produce an operational tool.

Keyword Codes: D.2.5; D.2.2

Keywords: Software Engineering, Testing and Debugging; Software Engineering, Tools and Techniques

1. INTRODUCTION

Software testing is a crucial component of the software development process. The traditional manual approach to testing is both time consuming and error prone. It is the aim of the present study to investigate into the feasibility of automating one aspect of testing, namely coverage testing. This paper describes the prototype of an automated coverage testing tool for the 4GL NATURAL[®] running under both the MS-Window and the UNIX[®] environment. Minor modifications is needed to extend the testing functions for other environments.

In coverage testing [1], the goal is to execute all the possible test paths for a given program. The selection of a particular test path is determined by conditions of the decision constructs in the program under test. Exhaustive testing of all the paths by manual methods is normally impractical as the number of distinct test paths is usually very large. The automated coverage testing procedures chosen is based on a test path selection technique proposed by Thomas J.

McCabe[2,3]. In McCabe's Baseline method, the total number of test paths is first determined by computing the cyclomatic complexity [4,5]. The other steps are:

- derive the flow graph from analysis of the source program under test;
- select a baseline path that preferably satisfy the following criteria:
 - perform a major function in the program and not an error path, and
 - intersects a maximum number of decisions in the flow graph;
- select second test path by flipping the condition of the first decision statement on the baseline path while keeping the other conditions of the original path unchanged;
- the third, fourth and subsequent test paths can be generated by flipping the conditions of the second, third .. decision statements etc.

There are three stages in the testing. The first stage is to compute the cyclomatic complexity and to obtain the flow graph for a given program module. The second stage is to select a test path. The third stage is to generate the test data so as to execute the selected test path. Only one set of input data for all the test paths is used instead of generating a large number of test data sets for all the test paths. The path selection is affected by automatically substituting 'true' or 'false' in the conditions of the decision statements as required by the baseline method.

This paper is organized into four sections. Section two describes the testing procedure. Section three gives the functional components of the system and provides a sample execution summary report. Section 4 concludes and identifies areas for further developments.

2. TESTING PROCEDURE IMPLEMENTATION

The automated testing procedure consists of six steps is depicted in the diagram below:

Figure 1. Testing Procedures

NATURAL[®] source code is assumed to reside on a SUN SPARC workstation. The six steps are:

1. Source code is downloaded to a PC platform under MS-Window from UNIX.
2. The cyclomatic complexity and flow graph are obtained.
3. McCabe's Baseline method is used to select the appropriate test path by modifying conditions of decision statements in the source program.
4. A copy of the source with selected test path is uploaded to UNIX.
5. The source is compiled and executed on the SUN SPARC.
6. Statistics and run-time errors are recorded and downloaded to MS-Window for analysis and storage.

Another test path is then selected and the whole process is repeated until all the test paths have been exhausted. It is important to ensure that only one copy of the source program is kept in the system. The same set of test data will be used for all the test paths. A summary report of all errors is provided at the end of the testing.

3. FUNCTIONAL COMPONENTS OF THE AUTOMATED TESTING TOOL

The automated coverage testing tool besides selecting test paths has to collect statistical and error information. The tool has six major functional components: Lexical Analyzer, Flow Graph Generator, Test Path Generator, Test File Generator, File Transfer, and Error Report Analyzer.

◆ Lexical Analyzer

The lexical analyzer captures the NATURAL source code information and convert source statements into tokens. The major task of the lexical analyzer is to derive flow graph information from the tokens. The analyzer therefore focuses on capturing conditional constructs such as IF..., REPEAT..., DECIDE... etc. The information stored for each conditional construct include:

- token_type - the token value for the conditional construct ;
- start_line - the starting line number of the conditional construct in the program;
- end_line - the line number for the next construct.

The information is fed into the flow graph generator.

◆ Flow Graph Generator

The flow graph generator is capable of computing the cyclomatic complexity as well as deriving the flow graph for the program under test. The flow graph provides a visual means of showing the flow control structure of the program. A sample flow graph is given in figure 2 with the associated cyclomatic complexity $V(G)$ which is 17 in this case.

Each node in the flow graph can represent one of the four different classes of constructs for NATURAL[®]. They are:

- decision statements - e.g. IF..THEN, DECIDE ON;
- loop statements - e.g. REPEAT..UNTIL, READ, FIND;
- end statements - e.g. END-IF, END-DECIDE, END-REPEAT;
- sequential statements.

Figure 2. Sample flow graph

In the construction of the flow graph, input characters are arranged in the same order as they appear in the original source program. One complication arises as a result of the nesting nature of conditional constructs. Backtracking is normally required to locate previous nodes. Two separate data structures are used in the construction of flow graphs. They are the node information table and the edge information table.

◆ Test Path Generator

The test path generator generates all the test paths based on the Baseline technique of McCabe. The path selection technique comprises of three phases. The first phase is to create a look-up table for all branching nodes. The second phase is to invoke the algorithm for selecting the baseline path. In the final phase, a path selection algorithm is invoked to select the different test paths in accordance with the baseline technique. Each test path can be displayed. Figure 3 is a sample of the flow graph with the test path highlighted.

The nodes falling within the test path is displayed at the bottom of the screen.

◆ Test File Generator

The test path generator creates test files which are to be used in the coverage testing. Each test file corresponds to the enforcement of execution of a particular test path. The module examines each decision statement of the concerned path and by substitutes the conditions by either an always 'true' or 'false' expression as deemed necessary.

Figure 3. Sample test path applying on the flow graph

◆ File Transfer Software

The uploading and downloading of files between UNIX and PC are carried out with the help of typical FTP programs. Both operations are automated through executing upload and download script files.

◆ Error Report Analyzer

The module resides on the PC platform and provides analysis and summary of execution errors in the different test runs. The main report from the analyzer is an execution summary report. The report contains a line by line error summary of the test program. The format of the report is as given in figure 4.

Execution Summary Report

Total Number of Test Paths =32

Source Statement Number	Last Tested #test	Number of items exercise	Error Message if any
1	8	5	Nil
2	2	2	divide by zero

Figure 4. A Sample Execution Summary Report

4. CONCLUSION AND FUTURE DEVELOPMENT

A prototype for automating the coverage testing process has been implemented and tested with 35 NATURAL programs of various sizes. Overall it has been found to be extremely useful and can in effect be used to reduce some of the workload of program testers. There are still many problems needed to be resolved before the tool can become operational. On the more fundamental issues:

- The arbitrary way of setting the conditions in the decision constructs is debatable. A proper way is to derive input data from output requirements for each test path.
- The lexical analyzer at present has difficulty in recognizing multi-words keywords such as AT END OF DATA.. etc. (keywords are also embedded).

Other issues include:

- Line crossing in flow graph is difficult to handle and should be avoided.
- The lexical analysis is a time-consuming process because of the large vocabulary of NATURAL.

Future work should look at ways of deriving input test data automatically for each test path. The ability to detect features such as the existence of dead code will be most useful.

ACKNOWLEDGEMENT

We would like to thank Mr. Wilson Hau for his assistance on this paper.

REFERENCES

1. Beizer, B.: 'Software System Testing and Quality Assurance', (Van Nostrand Reinhold, 1984)
2. McCabe, T.J. and Schulmeyer, G.G.: 'System Testing Aided by Structured Analysis - A Practical Experience', *IEEE Proceedings of COMPSAC*, pp.522-8, 1982
3. McCabe, T.J.: 'Structured Testing: A Testing Methodology Using the McCabe Complexity Metric', NBS Special Publication, 1982, pp.82-119
4. McCabe T.J.: 'A Complexity Measure', *IEEE Trans. on Software Engineering*, December 1976, pp.308-20,
5. McCabe, T.J., Young, L.F., Calybaugh, K.W., and McManus, J.: 'Design Basis Paths: A Complexity Driven Design Inspection Methodology', *Proceedings of the Total Systems Reliability Symposium*, 1983, pp.67-72