Sufficiently Large Number for Testing C++ Standard Template Library Modules

Ponrudee Netisopakul
Ponrudee@it.kmitl.ac.th

Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang
Bangkok, 10520 Thailand

Abstract

One of the most crucial questions in software testing is when to stop testing. The problem was proved to be an undecidable problem in general. However, for a set of collection programs in C++ called standard template library (STL), we are able to show that there exist a sufficiently large number N associated with the test data set size of each module, such that testing with a data set size larger than N does not reveal more faults in the module under test. This paper presents the concept, theorem, and the experimental results that confirm the finding of the sufficiently large number.

Keywords: Sufficiently large number, Data Coverage Testing, C++ Standard Template Library, Automated Test Generation, Testing of Collection Programs.

1. Introduction

It is well-known that many fundamental issues in software testing are undecidable. One famous example of an undecidable problem in an area of computational theory is the halting problem. Any problem to which the halting problem can be reduced is also an undecidable problem. In 1976, Howden [7] has proved the test selection problem to be undecidable. That is, although we know that a reliable test set exists for each program, no algorithmic method exists for constructing such a set for an arbitrary program.

The current trend toward software testing approach is to utilize automated testing. Automated testing employs automated input generation to generate a large set of test inputs for testing a program. The size of possible test inputs is infinite; therefore, it is almost immediate that criteria

for stopping test must be provided. However, this is an undecidable problem for an arbitrary program. The current practice has been focused on satisfying some specific structural features of a given program. For example, the criterion for statement coverage testing is to execute every statement in the given program. The criterion for branch coverage testing is to execute every branch in the given program. The problem with this approach is that no structural coverage strategy, even path coverage can guarantee the adequate testing, i.e., one correct output does not mean that every output is correct [2]. However, one example where automated input generation proves useful is the set of programs from the C++ standard template library, in short, STL programs, also sometimes called collection programs.

Although structural coverage strategy is still inadequate criteria for testing STL programs, we have utilized data coverage testing strategy [12] [13] which combine both white-box and black-box testing techniques to construct useful test set. The specific features of STL programs allow this technique to determine the stopping criteria of each program under test. Section 2 will depict the importance features of STL programs. Section 3 will describe the concept of data coverage testing technique. Section 4 and 5 will show the experimental results for some STL programs.

2. The C++ Standard Template Library (STL programs)

STL programs work with a collection of the same data type stored in container classes such as arrays, vectors, lists, queues, sets, etc. One important input parameter for this set of programs is the size of the input containers. The larger the container size, the more input combinations can be generated. In general, for a vector container of size n with k possible values, there are at most k^n distinct combinations of container contents.

Moreover, each program has at least two input parameters to specify the sub-range of the container under test called a *test window*, represented by the range [*first*, *last*). The program starts working on the element pointed to by the iterator *first*, and stops at the element before the one pointed to by *last*. Note that for each container size, test windows of size 0 can be generated by setting the parameter *first* to equal to the parameter *last*. When a test window size is smaller than a container size, the tester must also ensure that the content outside the test window has not been changed by faults in the program under test. The input generation must be able to generate various sizes of test windows within a container. This factor produces test input combinations

which grow as $O(n^2)$, to be considered a factor with the k^n growth. Obviously, if the container size is large, it will take too long to test all input combinations. On the other hand, if we know that testing up to a fixed small container size N is sufficient, then the testing could be stopped much earlier. Intuitively, we called this N a *sufficiently large* number.

Another important question for automated testing concerns how to generate test inputs. A *test model* for a given program specifies the details of how to conduct automated testing for the program under test. For example, for STL collection programs, a test model must specify how to generate the content of the input container, how to generate sub-ranges within the input container and how to generate an oracle for the program. Given a program and a test model, we will show that there exists a *sufficiently large* test data set size N, such that testing with data set size larger than N does not detect more faults.

Scalability is an important concern in any software testing technique. This paper targets container class libraries, where programs are often complex, but are typically relatively short. For example, in the C++ STL, many of the functions are well under 100 lines. The same is true of the containers in the Java libraries.

3. Theory of Data Coverage Testing

3.1 Definition of Sufficiently Large

Given a test model and a program under test, data coverage testing systematically generates increasing numbers of test inputs. For example, a test model for the *collection* replace program specifies two possible values to be generated in each position of the input container: an *old value*, which will be replaced by a *new value*, and a *controlled value* which must remain unchanged after the execution of the replace program. For a container of size n, 2^n different container contents will be generated for testing the replace program. The research goal of data coverage is to find an upper bound or a *sufficiently large* N associated with the container size of a given program.

Definition 1 *Given a program that works with containers, i.e., a collection program, assume data coverage testing and some set of assumptions related to that program; the input container of size N is* **sufficiently large** *if for every container of size $N' > N$, any test input of size N'*

cannot reveal any new fault other than all the faults previously revealed by test input of size 0, I,.., N.

We have developed a methodology to obtain this required N associated with a given program and theoretically proved that N obtained by this method is indeed sufficiently large. The assumptions mentioned in Definition 1 will be given in section 3.2.

3.2 Data Coverage Concepts

The first concept is a method of partitioning iteration loops into several levels introduced by Paige (1975) [10]. It partitions a program graph to capture the effect of an iteration loop at each level. Each path or loop in the program is partitioned into a *level-i path* or *level-i loop*. A *level-i* path is a simple path or a *simple loop* which begins and ends on nodes of lower level. (A simple path is a path in which each node and edge is traversed only once; a simple loop is a simple path with the same first and last node). A level-0 path begins with the entry node of the program graph and traverses through nodes without a loop until it reaches the exit node of the program. A level-1 path/loop begins and ends with nodes of a level-0 path, but nodes in between are new nodes never traversed as a level-2 path/loop begins and ends with level-1 nodes but all nodes in between are new nodes never traversed before. In other words, level-0 is a direct path from entry to exit without going through any loop. Level-1 is the first level loop; level-2 is a nested loop inside a loop of level-1 , and so on. Paige proved that the union of all level-i paths/loops contains all nodes and edges in the program graph.

The second concept is closely related to the *equivalence partitioning (EP)* concept introduced by Myers (1976) [9]. The idea of EP is to partition the input domain of a program into a finite number of equivalence classes so that a test of a representative value of each class is equivalent to a test of any other value from that class. Note that the process of identifying the equivalence classes is largely an heuristic process. Instead of adopting the entire EP concept, we adapted the concept to *partial input* classes and *partial output* classes, both related to the concept of level-i path just described. A *partial input* of a level-i path/loop is a representative of an equivalence class of inputs at the beginning of the level-i path/loop. A *partial output* of a level-i path/loop is a representative of an equivalence class of outputs at the end of the level-i path/loop. In other words, a level-i path/loop takes a partial input and produces a partial output. Some important

partial input/output classes for C++ STL programs are defined using the relative position of iterators *first* and *last*. The partial input/output classes of *first equals last* define the termination condition of many programs, while partial input classes of *first* < *last* define the operation within the loop.

The snapshot of a program that takes a partial input, executes a level-i path/loop, and produces a partial output can be represented by a triple vector (I_x, L_y, O_z), where I_x stands for a partial input of class x, L_y stands for a level-i path/loop y, and O_z stands for a partial output of class z. Because the general behavior of collection programs, which move to the next position after each iteration, corresponds to this triple vector, we call this triple vector a *move* as defined next.

Definition 2 A move (I_x, L_y, O_z) *of a given collection program is an execution of a level-i path/loop L_y of the program taking a partial input I_x and producing a partial output O_z.*

The concept of a move is essential because it allows us to capture a discrete unit of program sub-behavior. Moreover, two sub-behaviors of a program can also be compared using this concept.

Definition 3 *If two moves of a given collection program execute the same level-i path taking the same partial input and producing the same partial output, both moves are* **identical**.

From the testing viewpoint, if two moves are identical, then testing either of them is the same with respect to fault detection; only one of them is required to be tested. This observation motivates the following two definitions.

Definition 4 A base move set *of a given collection program is a set of moves with properties:*

1 . No two moves in the set are identical.

2. The union of all moves in the set constitutes all moves of the given program.

Definition 5 A repeated move set *of a given collection program is a set of moves such that for every move in the set, there is a move in the base move set that is identical to this move.*

The general idea is to find two sets of sub-behaviors for the given program. The first set contains a number of unique behaviors called *base moves*. The second set, called *repeated moves*, contains a number of repetitive behaviors, each *identical* to one in the *base move* set. Once both sets are identified, only the moves in the *base move* set need to be tested. Testing the *repeated move* set does not yield more faults.

Although the data coverage concept has been specifically applied to the collection programs, it can also be applied to many other programs; yet the data coverage concept cannot be applied to

every program. For example, we may not able to establish a finite number of moves in a base move set for some programs. In order to use the data coverage concept, the following assumptions related to the given program are needed. A violation of these assumptions may preclude the establishment of a sufficiently large N.

1. The program is deterministic.
2. The program has no uncontrollable random variable.
3. The number of different values of elements is finite and can be bounded.
4. The oracle for the solution of the program can be automatically executed.
5. The size of the input domain must be fixed during an execution.

3.3 Implication of the Assumptions

In order to construct a finite base move set, assumptions 1 , 2 and 3 must hold. Since lemma 1 and theorem 1 will conclude that we can partition all moves of the given program into two sets: a finite base move set and a repeated move set, assumptions 1, 2 and 3 must hold for them. For the rest of the paper, all assumptions 1 to 5 above should hold; assumptions 4 and 5 are needed for automated testing. Because of limited space, proofs are not provided here.

lemma 1 *Given a collection program that can be partitioned into two sets: a base move set and a repeated move set, the base move set is finite and all the moves in the base move set need to be thoroughly tested.*

Theorem 1 *Given a collection program that can be partitioned into two sets, a base move set and a repeated move set; if the base move set has been thoroughly tested, then testing the moves in the repeated move set can reveal no more faults. Therefore, it is sufficient to test only all moves in the base move set.*

4 Experimental Designs

4.1 Fault Seeding Procedure

We asked a number of volunteers to help create faults to seed into the three programs. Each volunteer was given one or more original correct programs. We asked the volunteers to create as

many faulty versions of the original program as possible, based on their own previous experience in programming. The experiments were conducted with three groups of volunteers: Group 1 was composed of students taking a graduate software engineering class. Group 2 was composed of six individuals selected by the author; these individuals did not know each other. The characteristics of Group 2 were varied; one might be an expert in C++, while another might be just a novice C++ programmer. Group 1 and Group 2 had neither specific knowledge of the software testing method used, nor how the experiment would use their faulty programs. Moreover, if the same faulty version was already provided by Group 1, it was excluded from Group 2. Group 3 was just composed of one individual: a third-party software testing expert who provided a very systematic fault seeding procedure.

4.2 Test Procedure

Steps in data coverage testing are:

1. Design a test model for each program.
2. Determine the set of possible path at each level by analysis of control flow graph.
3. Determine a set of partial inputs, I, and partial output O.
4. Derive a base move set by generating test input for container size 0, 1, 2, ... and so on. Add each unique move to the base move set until no more can be added because either:
5. Note the largest container size, **n**, used in constructing the base move set. This is the desired sufficiently large number N for the given program.

	lines	N	fault seeded versions
replace	7	4	33
partition	21	4	69
sort	65	35	234

Table 1: Details of STL programs used in the experiment.

Note that the required N for the sort program is considered very large, i.e., the test driver for N = 35 would generate $35^{35} \times O(35^2)$ test inputs. Therefore, we decided to use reduced test set for the sort experiments instead of the full test set.

5 Experimental Results

Experiment confirms the existing sufficiently large number

Figures 1 to 3 show experimental results for seven experiments. Note that the first two experiments involve the replace program with the faulty versions provided by two different groups of volunteers. Experiments 3, 4, and 5 involve the partition program with the faulty versions provided by three different groups of volunteers. The last two experiments involve the sort program with the faulty versions provided by two different groups of volunteers.

Figure 1 illustrates the result of experiments 1 and 2. Both experiments show that for the replace program, no additional faults are detected after the container size $N = 3$ are reached. Recall that the sufficiently large N for the replace program is N=4.

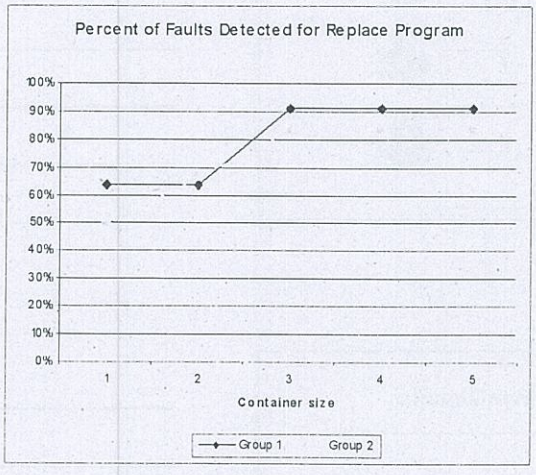

Figure 1: Percent of Faults Detected for Replace Program

Figure 2 illustrates the result of experiments 3, 4, and 5, i.e., the experiments on the partition program. For all 3 experiments, no additional faults are detected after the theoretical $N = 4$.

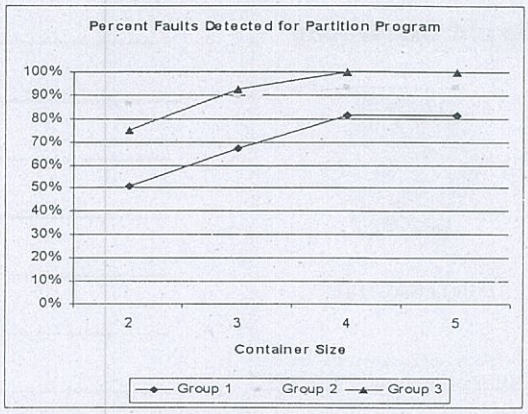

Figure 2: Percent of Faults Detected for Partition Program

Figure 3 illustrates the result detected for the sort experiments 6 and 7. The percentage of faults detected starts to reach a plateau after N = 36. However, some new faults continue to be detected until n reached 50. Although most of the faults were detected at the theoretically required N = 36, these experiments demonstrated that when the testing was not complete, a few faults could be missed at the required N.

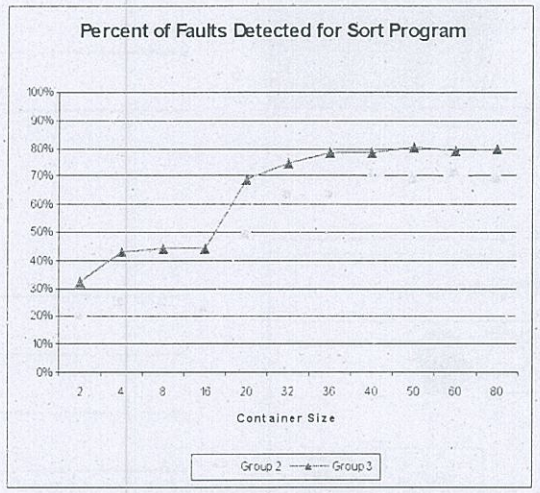

Figure 3: Percent of Faults Detected for Sort Program

6. Conclusions and Future Research

Although the term data coverage may not have been used before, we can identify a number of previous research results which combined information from both program specification and program implementation to aid the testing process. Richardson and Clarke (1985) [14] used a symbolic evaluation technique to generate test data, and later integrated it into a partition analysis system, combining information from both specification and implementation to derive test inputs.

More recent work includes that of Homan et al. (1999) [5] and Ball et al. (2000) [1]. Homan et al. (1999) formalized one type of data coverage measure and used it to generate test inputs for three C++ STL programs. Ball et al. (2000) pursued the approach by Homan and Strooper (1997) [4] implementing a state generation algorithm to generate test inputs for testing a container class implemented with a red-black tree data structure. Although the theory of section 3 was not used in that work, the experimental results did show that beyond small set sizes, test effectiveness did not improve. In particular, tests based on sets with fewer than 20 elements achieved 100% statement coverage and100% feasible path coverage, and revealed all seeded faults in a complex implementation of more than 600 lines. Although all previous work has introduced a number of data coverage testing techniques, none of them extensively addressed the issue of when to stop testing. Many experiments have been conducted to determine the effectiveness of structural coverage. These included Frankl and Weiss (1993) [3] and Hutchins et al. (1994) [8].

One of the most interesting questions in software testing is when to stop testing. Different testing techniques give different guidelines for stopping the testing process. Many researchers emphasize a guideline based on the structure of the program. One of the commonly used structural coverage measure is statement coverage, which suggests that the testing stops when all statements in the program have been executed at least once. Another guideline employs only the program specification to derive the tests. Our research uses a combined testing technique called data coverage, which utilizes information from both program specification and program implementation to find a satisfactory stopping point

This study was conducted with three collection programs from the set of C++ STL programs. It was shown that given a specific collection program, we can find a sufficiently large container

size N for testing that program. A number of experiments have been conducted that confirm this theoretical finding, and were consistent with the N obtained.

A number of interesting future studies could be done with data coverage testing. First, a definition of sufficiently large is now defined for the class of programs called collection programs. It would be interesting to study how the concept can be extended and applied to more general programs.

Another realistic and interesting study is specification-based data coverage. In this case, data coverage testing are generated only from the specification. Unlike program-based data coverage where the source code of a program under test can be analyzed before testing, specification-based data coverage must be robust enough to test many different implementations of a program including those that have not yet been written. Therefore, specification-based data coverage testing is a much more difficult research problem. For example, a number of restrictions must be developed to constrain the programs that can be tested in this way.

7. References

[1] Ball, T., Homan, D., Rusky, F., Webber, R., and White, L. (2000), 'State generation and automated class testing', Software Testing, Verification and Reliability, **10**, 1 49- 70.

[2] Fenton, N., and Pfleeger, S. L., (1997) 'Software Metrics: A rigorous & Practical Approach', 2nd Edition, PWS Publishing Company.

[3] Frankl, P. G., and Weiss, S. N., (1993), 'An experimental comparison of the effectiveness of branch testing and dataflow testing', IEEE Trans.on Software Eng., **19(8)**, 774-787.

[4] Hoffman, D., and Strooper, P., (1997), 'ClassBench: a framework for automated class testing', Software Practice and Experience, **27(5)**, 573-597.

[5] Hoffman, D., Strooper, P., and White, L. (1999), 'Boundary values and automated component testing', Software Testing, Verification and Reliability, **9**, 3-26.

[6] Horgan, J. R. and Mathur, A. P., "Software Testing and Reliability", Chapter 13, Handbook of Software Reliability Eng., Ed M. Lyu, McGraw-Hill, 1996.

[7] Howden, W.E. (1976) 'Reliability of the path analysis testing strategy', IEEE Transaction of Software Engineering, **SE-2** (3), 208-215.

[8] Hutchins, M., Foster, H., Goradia, T., and Ostrand, T., (1994), 'Experiments on the effectiveness of dataflow-based and controlflow-based test adequacy criteria', Proc. 16th Int. Conf. on Software Eng., Sorrento, Italy, May 1 994, 53-66.

[9] Myers, G. J. (1976), 'The art of software testing', John Wiley and Sons, New York.

[10] Paige, M. R. (1975), 'Program graphs, an algebra, and their implication for programming', IEEE Trans. on Software Eng., **1 (3)**, 286-291.

[11] Netisopakul, P., White, L. J., Morris, J., (2002), 'Testing Efficiency: Statement Coverage, Random Testing, and Data Coverage Testing", International Conference on Practical Software Quality Techniques and International Conference on Practical Software Testing Techniques (PSQT/PSTT), March 4-8, 2002, New Orleans, Louisiana.

[12] Netisopakul, P., White, L. J., Morris, J. and Hoffman, D., (2002) 'Data Coverage Testing of Programs for Container Classes', Proc. of the Int. Symposium on Software Reliability Engineering (ISSRE-2002), Annapolis, MD, Nov 12-15, 2002, 183-194.

[13] Netisopakul, P., White, L. J., Morris, J., (2002), 'Data Coverage Testing", Proceeding of Asia-Pacific Software Engineering Conference, December 4-6, 2002, Gold Coast, Queensland, Australia. 465-472.

[14] Richardson, D. J., and Clarke, L. A.(1985), 'Partition Analysis: A method of combining testing and verification', IEEE Trans. on Software Eng., **11(2)**, 1477- 1490.

[15] Roper, M., Wood, M., and Miller, J.(1997), 'An empirical evaluation of defect detecting techniques', Information and Software Tech., 39, 763-775.

[16] White, L. J., (1987), 'Software Testing and Verification', Advances in Computers, Vol 26, 335-391.