larger as the array size increases.

The experimental results indicate that as the state space becomes smaller, the solving time can be reduced more dramatically. For instance, let us consider Table 6 which shows average solving times and their ratios for TR3 with respect to various array sizes and integer domains.

The column `ratio1` lists the ratios between solving times for `dom1` and `dom3` and the column `ratio2` lists the ratios between solving times for `dom1` and `dom2`. Remind that `dom3` is four times smaller than `dom1` and is two times smaller than `dom2`. The largest ratio is about 68 (≈726365/10644). It occurs when the array size is 20 and the integer domain is reduced from `dom1` to `dom3`. When compared to `ratio2`, it shows the significant reduction in solving time.

We can gain further insight from this study. The state space should be reduced without losing optimal solutions. For instance, consider TR1 which is unsatisfiable. TR1 is unsatisfiable not only on `dom1` but also on smaller domains `dom2` and `dom3`. It indicates that if we draw some results on small domains, then the same results will be possibly drawn from larger domains.

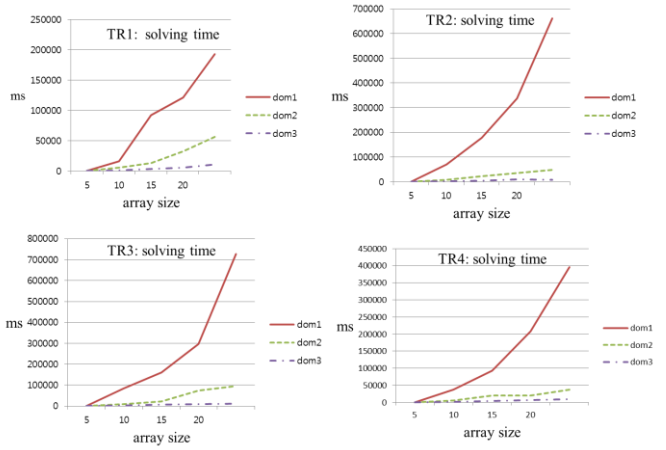

Figure 22. Experimental results

Table 6: Ratios between Solving Times for TR3

array size	dom1	dom2	dom3	ratio1	ratio2
5	85302	8644	1706	50	10
10	161087	22710	6234	26	7
15	295824	73449	9172	32	4
20	726365	94749	10644	68	8

In facts, our method relies on *the small scope hypothesis* in traditional software testing [13, 14]. The small scope hypothesis states that we can detect a high proportion of errors by testing a program for all test inputs within some small scope. This suggests that we can save a lot of testing efforts to test a program exhaustively for some bounded small scope instead of deliberately selecting test inputs from a larger one. Thus, it is important to find a way in which the state space can be reduced while preserving optimal solutions.

RELATED WORK

A large portion of test techniques can be divided into two approaches: path-oriented approach and goal-oriented approach [15]. The path-oriented approach demands an entire path going through a target and tries to find test input on which the given path is executed. It often uses symbolic execution. Symbolic execution derives a system of algebraic constraints from a given path in terms of symbols representing any input values that can traverse the path [16]. Solutions to the system of algebraic constraints become test data that cause the execution of the path.

Yan and Zhang proposed a path-oriented method for basis path testing [3]. Their work was motivated by existing basis path testing methods. Poole's method and the baseline method are well known methods used for generating basis paths from a given program [4, 5]. The methods build the control flow graph of a program to find its basis paths. As a result, basis paths frequently contain infeasible paths. Based on the observation, the method proposed by Yan and Zhang continues to generate paths until the number of feasible basis paths reaches the cyclomatic complexity of the program. However, their method suffers from the same problems as the path-oriented approach. When testing a program containing loops, the number of paths to be explored can be possibly unbounded. Unfortunately, a large portion of the paths is infeasible. Clearly, this wastes a lot of search effort. Even if their method can find desired number of feasible paths, furthermore, some of the paths are redundant. For instance, the method could continuously generate paths of the form `<s,(N1,N2,N3,N4,N5,N6)+,N1,e>` where '`+`' denotes multiple occurrences of the sequence within parentheses for the example program in Figure 21. Irrespective of how many times the loop iterates, all paths of the form are infeasible. Generation of those paths causes a large number of attempts before the search procedure terminates and a lot of effort can be wasted. Furthermore, the method would continue the search process until the number of generated feasible paths reaches the cyclomatic number of the program. It could find an additional (feasible) path satisfying TR2, TR3, or TR4 rather than TR1. If the method generates a path of the form `<s,(N1,N2,N3,N4,N6)+,N1,e>`, the path would be taken into account as an element of the basis path set even if it satisfies TR2.

The goal-oriented approach aims to produce inputs which execute the given target without a specific path. Many goal-oriented techniques can be categorized as dynamic because they require the execution of the program. Dynamic test data generation methods rely on distance function to discriminate between candidate tests in terms of the cost required to

achieve the test goal. One typical example of the goal-oriented approach is the chaining method presented by Ferguson and Korel [17]. The chaining method takes use of data flow dependency to identify statements affecting the branch predicate at which the branch function cannot guide the search and attempts to find inputs that cause those statements to be executed. Because the path taken is determined on the fly, it is very difficult to predict which path will be taken in advance.

Some of the goal-oriented approach us constraint solving techniques for test generation. Gotlieb et al presented a goal-oriented testing method which transforms a program into a constraint system to take advantage of constraint solvers [18]. Unlike the proposed method, however, the method is tightly coupled with the statement coverage or branch coverage criteria.

We can frequently encounter software testing techniques using evolutionary algorithms. Ghiduk proposed an interesting approach to basis path testing using a genetic algorithm, in which the length of each chromosome varies from iteration to iteration according to the change in the length of the path [19]. The approach focused on generation of the basis paths. However, Ghiduk did not describe how to discover test inputs to traverse them.

CONCLUDING REMARKS

We presented a SAT-based test generation method that follows the goal-oriented testing approach for basis path testing. A large proportion of existing test data generation methods has focused on the statement or branch test coverage criteria which require traversal of a selected program point (statement or branch). Unlike the statement or branch test coverage criteria, basis path testing requires multiple program points to be traversed to satisfy test requirements rather than a single program point while satisfying certain constraints among those program points. SAT solving techniques allow us to effectively specify constraints in test requirements. We discussed how to transform test requirements for various test criteria including basis path testing into constraints for SAT solving. One notable benefit of our SAT-based testing is to generate feasible basis paths of optimal lengths. The detection of the infeasibility of a given basis path can also be done as early as possible with the SAT solver. We also presented a procedure for the case where a basis path is infeasible. The procedure can possibly generate another feasible path which is linearly independent with the existing set of basis paths. The proposed method enables us to save a lot of testing efforts by avoiding or reducing trials of generating redundant basis paths.

ACKNOWLEDGEMENTS

This research was financially supported by Hansung University.

REFERENCES

[1] T.J. McCabe, “A complexity measure”, IEEE Trans. Software Eng. vol. SE-2, no. 4, (1976), pp. 308–320.

[2] P.A.V. Zhu and J.H.R. Hall, “Software unit test coverage and adequacy”, ACM Comput. Surv. vol. 29, no.4, (1997), pp. 366–427.

[3] J. Yan and J. Zhang, “An efficient method to generate feasible paths for basis path testing”, Information Processing Letters, 107, (2008), pp. 87-92.

[4] J. Poole, “A method to determine a basis set of paths to perform program testing”, (NISTIR 5737), Tech. rep., Department of Commerce, NIST, Nov. 1995.

[5] A.H. Watson and T.J. McCabe,“Structured testing: A testing methodology using the cyclomatic complexity metric”, NIST Special Publication, (1996), pp. 500-235

[6] L. Gregory, “Path testing”, http://www.cs.swan.ac.uk/~csmarkus/CS339/dissertations/GregoryL.pdf, (2007).

[7] E. Torlak and D. Jackson, “Kodkod for Alloy users”, Proceedings of First Alloy Workshop, Portland, (2006).

[8] I. S. Chung, “CEGPairGen: An automated tool for generating pairwise tests from cause-effect graphs”, International Journal of Software Engineering and Its Applications, vol. 9, no. 1, (2015), pp. 53-66.

[9] I. S. Chung, “Using boolean satisfiability solving for pairwise test generation from cause-effect graphs: comparison of three approaches”, International Journal of Software Engineering and Its Applications, vol. 9, no. 9, (2015), pp. 65-78.

[10] I. S. Chung and J. M. Bieman, “Generating input data structures for automated program testing”, Journal of Software Testing, Verification, and Reliability, vol. 19, issue 1, (2009), pp. 3-36.

[11] SAT4J, http://www.sat4j.org/.

[12] B. Korel, “Automated software test data generation”, IEEE Transactions on Software Engineering, vol. 16, no. 8, (1990), pp. 870–879.

[13] D. Jackson and C. A. Damon, “Elements of style: Analyzing a software design feature with a counterexample detector”, IEEE Transactions on Software Engineering, vol. 22, no. 7, (1996), pp. 484-495.

[14] A. Andoni, D. Daniliuc, S. Khurshid, and D. Marinov, “Evaluating the Small Scope Hypothesis”, Technical report, MIT CSAIL, (2003).

[15] J. Edvardsson, “A survey on automatic test data generation”, Proceedings of the Second Conference on Computer Science and Engineering, (1999), pp. 21-28.

[16] L. A. Clarke, “A system to generate test data and symbolically execute program”, IEEE Transactions on Software Engineering, vol. 2, no. 3, (1976), pp. 215–

222.

[17] R. Ferguson, B. Korel, "The chaining approach to software test data generation", ACM Transactions on Software Engineering and Methodology, vol. 5, no. 1, (1996), pp. 63-86.

[18] A. Gotlieb, B. Botella, and M. Rueher, "Automatic test data generation using constraint solving techniques", Proceedings of ACM ISSTA, (1998), pp. 53–62.

[19] A. S. Ghiduk, "Automatic generation of basis test paths using variable length genetic algorithm", Information Processing Letters, 114, (2014), pp. 304-316.