Performance Analysis of Metaheuristic and Constraint Programming Approaches to Generate Structural Test Cases

Neelesh Bhattacharya[1,2], Abdelilah Sakti[2,3], Giuliano Antoniol[1], Yann-Gaël Guéhéneuc[2], and Gilles Pesant[3]
[1] *SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada*
[2] *Ptidej Team, DGIGL, École Polytechnique de Montréal, Canada*
[3] *Quosséça Lab, Department of Computer and Software Engineering*
École Polytechnique de Montréal, Québec, Canada

I. Introduction

Structural test case generation has been carried out by various approaches in software testing. Metaheuristics and constraint programming approaches are two of the more important approaches used for generating structural test cases [1], [2]. However, both of these approaches have limitations, which prevent them to be used in various applications, like wireless communication and aeronautical engineering, because the problems in these areas involve variables with large domains and complex constraints.

A. Limitations of CP and Metaheuristic Approaches

Metaheuristics have proved relevant to generate test cases for feasible test targets (e.g., executing a statement in the program under test). However, in most cases, they get stuck in local optima and fail to prove that a test target is infeasible. This limitation is a major drawback considering that a program may contain a significant number of infeasible paths (which can be a part of the test requirement).

Constraint programming overcomes the limitations of metaheuristics but suffers in terms of execution time when the input domain is large. When the number of input variables of the problem to solve to generate test cases is large or when the constraints are too complex constraint programming takes a long time to reach a good solution.

B. Goal

To overcome the limitations of constraint programming and metaheuristic approaches and get the best of both worlds, we want to propose a way to combine both approaches and the order in which they should be executed. Combining both of the approaches would allow their use in the various applications where they could not apply before.

C. Problem

For combining both metaheuristic and constraint programming approaches, we must have sufficient information about properties of both these approaches. One such property is their performance. Performances can be compared in terms of execution time, coverage criteria (like branch and path coverage), number of fitness calculations required by an approach to reach a test goal, etc. In our case, we use the of number of fitness calculations required and execution time.

Comparison of the relative performance of constraint programming and metaheuristic approaches gives us the information with which to decide of the order in which the approaches would be executed on a sample code.

D. Solution

To compare their performance, we generate test cases to fire divide-by-zero exceptions in a sample code and compare the number of fitness calculations and execution time required by both of these approaches.

II. Empirical Study Design

A. GQM

Our goal is to compare the performance of constraint programming and metaheuristic approaches for generating test cases to fire divide-by-zero exceptions on a sample code. Our question is which is the most efficient approach to generate such test cases. We use the required number of fitness calculations and execution time as our metrics.

B. Independent/Dependent Variables

Our independent variable is the choice of the approach, either metaheutistics or constraint programming. Number of fitness calculations and execution time are the dependent variables. The input variable domains of the program under test vary in the range of $\pm$500 to $\pm$500,000.

C. Subject Program

We use a modified form of the code presented in [3] as our program under test (PUT). We characterize the PUT using five dimensions: the domain size of the program input parameters, the program size (number of lines of code), the level of nested conditional statements (number of nested conditional statements), existence of pointers in the program, and presence of function calls in the program. The characteristics are: -500000 $\leqslant$ Domain size $\leqslant$ 500,000; LOC $\leqslant$ 50; no pointers; no function call; nested-If=1.

D. Subject Target

We categorize the test targets into goal-oriented target (which aims to generate test case for a specific criteria in the PUT) or path-oriented target (which aims to cover as many paths as possible in the PUT according to a specific coverage criteria). The test target for our PUT is goal-oriented and, firing the divide-by-zero exception(s) in the PUT.

E. Procedure

We carry out the preliminary instrumentation of the PUT by adding sufficient guard conditions, so that the generated test cases fire divide-by-zero exceptions (targets for our program) in the PUT. Thus, we generate test cases for the PUT using both metaheuristic and constraint programming.

The program, after being instrumented by constraint programming, is put into the CP model. To generate test cases using the constraint programming model, we follow the approach proposed in [4]. This approach translates a whole program and its control graph flow into a constraint satisfaction problem (CSP). Solving this CSP, we can generate test cases corresponding to a given statement, branch, path, or cover a given test criteria.

We compare two different metaheuristic approaches (hill climbing and genetic algorithm) and random generation to choose the best in terms of number of required fitness calculations. For the hill climbing, we use three strategies (variable neighbourhood search, gaussian generated neighbourhood search, and two sets of gaussian generated neighbourhood search) and choose the best one out of them.

Based on the number of fitness calculations required (to reach the goal every time) by each strategy, we choose two sets of gaussian generated neighbourhood search strategy for hill climbing and then compared its perfomance with random and genetic algorithm approaches. The fitness function used for the analysis considers both the branch distance and approach level [5]. We used the GA framework jMetal to obtain the candidate solutions for genetic algorithm.

We choose the best metaheuristic approach and compare with constraint programming. Test cases are generated and the execution time required for generating test cases (which fires an exception) are compared.

F. Analysis

We report the result of the empirical study in terms of number of fitness calculations required (for metaheuristic and random approaches), execution time (for metaheuristic and constraint programming approaches) and input domain size. We also carry out the performance comparison for random and two metahauristic approaches (Genetic Algorithm and Hill Climbing) in terms of execution time. We perform a statistical analysis on the obtained data using box-plots.

III. Empirical Study Results

For the various input domain ranges, the genetic algorithm requires the least number of fitness calculations for generating test cases. Hill climbing performs poorer to the genetic algorithm but much better than the random approach, which performs reasonably well when the input domain is very small ($\pm$500) but takes too long time to generate test cases when the domain goes beyond $\pm$5,000. The result is similar when the approaches are compared in terms of execution time.

IV. Conclusion

Comparing the execution times required for generating test cases using constraint programming and genetic algorithm, we found that, for all the input domains, constraint programming performs better in terms of execution time. When the goal is to fire divide-by-zero exceptions, constraint programming performs better, in terms of execution time, than even the best metaheuristic approach. We thus conclude that when we will combine both the approaches, constraint programming should be executed before metaheuristics, because constraint programming reaches a solution faster than any other metaheuristic approach.

V. Future Work

The order of execution of the approaches can only be generalized when we get similar results after carrying out experiments with many programs of various levels of size and complexity. Further, other parameters, apart from execution time, should be considered when comparing both the approaches. In the future, we will conducting experiments with well known programs so as to generalize our conclusion. Further, we are working on combining both metaheuristic and constraint pogramming approaches in an effective way.

References

[1] E. Diaz, J. Tuya, and R. Blanco, "Automated software testing using a metaheuristic technique based on tabu search," in *International Conference on Automated Software Engineering*, 2003.

[2] A. Gotlieb, "Euclide: A constraint-based testing framework for critical c programs," in *International Conference on Software Testing Verification and Validation*, 2009, pp. 151–160.

[3] N. Tracey, J. Clark, and J. Mcdermid, "Automated test-data generation for exception conditions," *Software - Practice and Experience*, vol. 30, pp. 61–79, 2000.

[4] A. Sakti, Y.-G. Guéhéneuc, and G. Pesant, "Structural test case generation for combined criteria," in *International Conference on Tests and Proofs, Submitted*, 2011.

[5] A. Baresel, H. Sthamer, and M. Schmidt, "Fitness function design to improve evolutionary structural testing," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2002, pp. 1329–1336.