2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

A Heuristic Approach to Break Cycles for the Class Integration Test Order Generation

Miao Zhang, Jacky Keung, Yan Xiao, Md Alamgir Kabir, Shuo Feng
Department of Computer Science
City University of Hong Kong
Hong Kong
{*miazhang9-c, yanxiao6-c, makabir4-c, shuofeng5-c*}*@my.cityu.edu.hk, Jacky.Keung@cityu.edu.hk*

***Abstract*—It is a general objective to minimize overall stubbing cost when performing class integration test order generation. Existing approaches are unable to obtain a cost-optimal class test order, this is largely due to the lack of a comprehensive analysis on the factors that affect overall stubbing cost, i.e., the number of required test stubs and the corresponding stubbing complexity. To address this issue, we propose an approach called HBCITO (Heuristic approach to Break Cycles for the class Integration Test Order generation). Given a set of removed dependencies, a heuristic algorithm is employed to search for a near ideal set of class dependencies. Such dependencies break the same or greater number of cycles as the initialized dependencies but attract less stubbing cost. The experimental results show that HBCITO is capable of generating class test orders with significantly lower stubbing cost compared with other approaches.**

***Keywords*-integration testing; class integration test order generation; break cycles; stubbing complexity**

I. Introduction

In integration testing, deciding the order in which classes are tested is non-trivial due to cycles among class dependencies [1]. When cycles exist, test stubs are inevitable since the dependent class can be unavailable. Various class test orders cause different test stubs, and consequently, different overall stubbing cost. Class integration test order generation aims to devise an optimal class test order attracting the minimum stubbing cost.

Most of graph-based approaches are unable to determine an optimal class test order since their cycle-breaking strategies ignore two factors that impact the stubbing cost, i.e., the number of created test stubs and the corresponding stubbing complexity. This limitation also occurs in search-based methods due to the lack of sufficient guidance in the search process.

To minimize overall stubbing cost by fully considering the aforementioned two factors, we propose a heuristic approach to break cycles for the class integration test order generation called HBCITO. Inspired by evolutionary algorithms, we put forward exploration and exploitation methods to break cycles when performing class integration test orders generation. In our approach, a class dependency is first initialized to remove a part of cycles. Then exploitation is adopted to search whether there exist other class dependencies that break the same cycles as the initialized dependency but attract less stubbing cost. Similarly, exploration is presented to find better alternatives that break the same number of cycles as the initialized dependency, which can achieve a better population diversity. The above process is repeated until no cycles remaining, and consequently, a class test order is generated.

Compared with the existing search-based approaches, a benefit of our method is that it can minimize the search space for large-scale programs with hundreds of classes. Unlike the previous search-based methods consider the full permutation of all classes, only class dependencies that are involved in cycles are considered in HBCITO.

We evaluate HBCITO by conducting experiments on three large-scale programs (JHotDraw, jmeter and log4j3) and three benchmark programs (ANT, ATM, and DNS). The results show that HBCITO outperforms the competitors in generating class test orders with minimum stubbing cost. Meanwhile, HBCITO reduces the search space for six programs. The main contributions of this paper are as follows:

- A new encoding strategy is introduced to describe the solution for class integration test order generation, which minimizes the search space.
- A heuristic algorithm is proposed to balance the two factors (the number of test stubs and the corresponding stubbing complexity) that affect the overall stubbing cost.

The remainder of this paper is organized as follows. Section II introduces the background. We present our approach in Section III. The experiments follow in Section IV. Section V concludes this work.

II. Background

In this section the needed background concepts are presented, including the preliminary knowledge and a literature review on the class integration test order generation.

A. Preliminaries

In integration testing, the order in which classes are integrated and tested has a significant impact on the construction

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.00016

of test stubs.

A test stub is built for the source class to emulate the behaviors of an unavailable dependent class. In order to measure the stubbing cost for a test stub (i, j), stubbing complexity is put forward by Briand et al. [2]. It is calculated as the following:

$$SCplx(i,j) = [W_A \cdot \overline{A(i,j)}^2 + W_M \cdot \overline{M(i,j)}^2]^{1/2} \quad (1)$$

Attribute coupling and method coupling, counting the number of attribute accesses and method invocations, affect the stubbing complexity. They are normalized and weighted to convert into a single objective in (1), which helps to understand and measure the effort to construct a test stub. Supposing that classes in a program are integrated and tested based on the test order o, and all test stubs compose a set *Stubs*. The stubbing cost of the entire integration test is estimated by overall stubbing complexity (OCplx), which is calculated as follows:

$$OCplx(o) = \sum_{(i,j) \in Stubs} SCplx(i,j) \quad (2)$$

The aim of class integration test order generation approaches is to minimize the overall stubbing cost by devising an optimal class test order.

B. Related Work

Several graph-based [3] and search-based approaches [2] have been proposed to generate class integration test orders automatically.

Graph-based approaches first construct a directed diagram to describe class dependencies, then break all cycles in the diagram by removing edges, and finally integrate classes based on topologically sorted order. Kung et al. [3] first provided a general way to break cycles but did not present elaborate rules to remove edges. Based on Kung et al.'s work, Tai and Daniels [4], Le Traon et al. [5] and Briand et al. [6] proposed different ways to assign a weight for each edge. Weights measure the number of cycles in which edges are involved, and consequently, the edge with the highest weight is removed. Unlike the aforementioned graph-based approaches, Hewett [7] discarded the process of cycle-breaking, and adopted an incremental strategy to minimize the number of test stubs.

However, the number of test stubs is unable to completely measure the stubbing effort for a given class test order. Therefore, other factors that affect the stubbing effort were introduced. For instance, Hanh [8] considered both stub minimization and testing resource allocation when generating class test orders. Malloy [9] believed types of class dependencies affect the stubbing cost and assigned them different weights. Similarly, Bansal [10] introduced new dependencies that were omitted by Malloy. Abdurazik and Offutt [11] proposed nine kinds of coupling relationships between classes, and adopted cycles-weight ratio to remove edges.

For search-based methods, Briand et al. [2] first applied genetic algorithm in the class integration test order generation problem. Simulated annealing algorithm is adopted by Borner et al. [12]. Multi-objective optimization algorithms were introduced to this area, such as ant colony algorithm adopted by Vergilio et al. [13] and NSGA-II applied by Assunção et al. [14]. To improve the performance of existing evolutionary algorithms, Guizzo et al. [15] introduced a hyper-heuristic to choose evolution operators. Mariani et al. [16] adopted grammatical evolution to generate multi-objective evolutionary algorithms automatically. Reinforcement learning is applied by Czibula et al. [17]. For search-based methods, finding a class test order with satisfactory result can be a hard task due to huge search space for complex programs with hundreds of classes.

III. Approach

This section introduces HBCITO, a heuristic approach to break cycles for the class integration test order generation. The procedure of this approach includes five steps, which is presented in Figure 1. For a subject program, the inputs are the class dependencies and related coupling information (e.g., the number of attribute accesses and method invocations), which are obtained by static analysis.

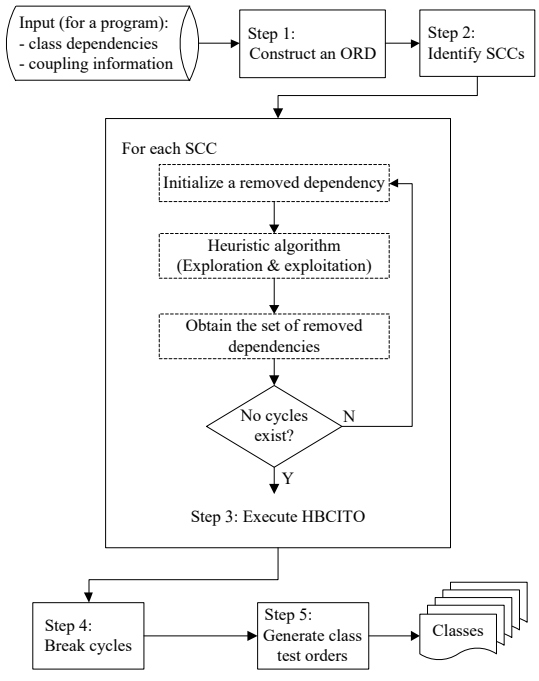

Figure 1. Procedure of HBCITO

First, an object relation diagram is created to represent class dependencies for the subject program. In such an

object relation diagram, each node represents a class, and a dependency from the source class to the target class is presented as a directed edge from the head node to the tail node. Stubbing complexity is calculated for each dependency based on the input attribute coupling and method coupling.

Then, the Tarjan's algorithm [18] is adopted to execute depth-first search for the object relation diagram and divides the diagram into multiple strongly connected components (SCCs). A strongly connected component containing more than one node is non-trivial. Cycles only exist in such a non-trivial strongly connected component. That's because all pairs of vertices in a cycle are connected, these vertices must be in the same strongly connected component. So we regard such strongly connected components as the smallest element to deal with, and identify all cycles in it.

For each non-trivial SCC, the HBCITO is applied. We first select a dependency as an initialization of removal, then apply the heuristic algorithm to search for an alternative that has the similar effect as removing the initial dependency but attracts less stubbing cost.

Our heuristic algorithm is inspired by the concept of exploitation and exploration from evolutionary algorithms. In evolutionary algorithms, exploration and exploitation are two main measures to search for optimal solutions. Exploration searches solutions in new regions of the search space [19], such as mutation and crossover operators in genetic algorithm, while exploitation searches solutions in the neighborhood of the current solutions [19], such as selection operator based on fitness.

Similarly, in our heuristic algorithm, exploitation is adopted to search another dependency or a set of other dependencies that break the same cycles with the initial choice but reduce the stubbing cost, while exploration searches some dependencies as new selection, which breaks the equal or greater number of cycles compared to the initial removed dependency.

The above mentioned heuristic algorithm is repeated until no cycles remain in the strongly connected component, and consequently, we obtain a satisfactory set of removed dependencies. After that, we move to the next strongly connected component, and apply HBCITO again. If all cycles are broken, test stubs are built for these removed dependencies.

Finally, we generate class test orders according to the following procedure. For each untested class, a dependent list is used to record its dependent classes. This dependent list is constructed based on the input class dependencies but omits the removed dependencies obtained from the last step. If a class does not depend on other classes, it is added into the test order directly. Once a class is integrated, it should be removed from the dependent lists of others. When the dependent list of a class is null, this class is added into class test order and removed from others' dependent list. This procedure is executed iteratively until all classes have been integrated and tested.

A. Heuristic Algorithm

Our heuristic algorithm is introduced in Algorithm 1, taking a set of current removed edges *currentEdges* as input, trying to find a set of removed edges *RemoveEdges* that break the same cycles, or at least the same number of cycles with *currentEdges*. Lines 1-3 are initialization procedure where setting *RemoveEdges* to empty, saving cycles that are broken by *currentEdges* , and counting their number. Then two functions exploitation and exploration are invoked to generate a set of removed edges that can replace the initial edges (lines 4-5). The results obtained by exploitation and exploration are compared, and the set with lower stubbing cost would be reserved (lines 6-10).

In the function exploitation, we search the edge with minimal stubbing cost for each cycle and add it into the alternative set (lines 12-14). if the new generated set attracts less stubbing cost, it would be returned (line 15).

Similarly, in the function exploration, we aim to break the same or greater number of cycles as *currentEdges* (lines 18-21). An alternative set is returned if its stubbing cost is less than that of *currentEdges* (line 22).

B. Search Space

In the conventional search-based methods, a vector of integers is used to represent a class test order, where each integer corresponds to a class. For a program containing N classes, the number of all possible class test orders is the factorial of N ($N!$). Hence, it is an arduous task to identify an optimal test order from such a huge search space containing tens of thousands of candidate solutions.

However, for our approach, a candidate solution for class integration test order only consists of removed dependencies. Supposing that only n dependencies are involved in cycles in a program, the number of all possible solutions in our method would be $2^n - 1$. It is proven that $N!$ is much greater than 2^N when N is larger than two according to mathematical theorems. What's more, we found that the number of dependencies that are involved in cycles (n) is fewer than the number of classes (N) for the large-scale programs in our experiments, i.e., $N > n$. Therefore, the search space of our method is smaller than that of the existing search-based methods, which can reduce the difficulty of searching for an optimal solution without deteriorating the final results.

IV. Experiments

A. Experimental Setup

We select two groups of programs to evaluate the performance of HBCITO. Table I shows the information of used programs, where columns 2 to 7 represent the number of classes, dependencies, cycles, dependencies that are involved in cycles, strongly connected components and lines of code, respectively.

Algorithm 1 Heuristic Algorithm

Input: a set of current removed edges *currentEdges*
Output: a set of removed edges *RemoveEdges*

```
1: RemoveEdges ← Φ
2: ListCycles = getCycles(currentEdges)
3: numOfCycles = getCyclesNum(currentEdges)
4: tempEdges1 = exploitation(ListCycles)
5: tempEdges2 = exploration(numOfCycles)
6: if tempEdges1.SCplx() > tempEdges2.SCplx() then
7:     RemoveEdges ← tempEdges2
8: else
9:     RemoveEdges ← tempEdges1
10: end if

11: function exploitation(ListCycles)
12: for each cycle c in ListCycles do
13:     find the edge in c with the minimal stubbing cost
        and add it into tempEdges1
14: end for
15: return tempEdges1 if it causes less stubbing cost
16: end function

17: function exploration(numOfCycles)
18: while totalCycles < numOfCycles do
19:     find the edge whose stubbing cost is less than that
        of currentEdges and add it into tempEdges2
20:     update totalCycles
21: end while
22: return tempEdges2 if it causes less stubbing cost
23: end function
```

Table I
INFORMATION OF USED PROGRAMS

Systems	Classes	Deps	Cycles	OverDeps	SCCs	LOC
JHotDraw	411	1680	225	199	318	78,150
jmeter	372	1252	729	147	309	32,882
log4j3	261	784	2173	242	156	15,665
ANT	25	83	654	41	14	4093
ATM	21	67	30	29	14	1390
DNS	61	276	16	23	53	6710

As shown in Table I, the number of dependencies that are involved in cycles is fewer than the total number of classes except for programs ANT and ATM. It is obvious that HBCITO can reduce the search space compared with conventional search-based methods because the number of all possible solutions for HBCITO is $2^n - 1$ (n is the number of distinct dependencies in cycles), rather than the factorial of N (N is the number of classes). We found the search space for ANT and ATM is also reduced after calculation.

The frist group contains three large-scale programs: jmeter (a load test tool) and log4j3 (a log tool for Java) are from SIR[1] (Software-artifact Infrastructure Repository), and JHotDraw is a 2D graphics framework used in the literature [14]. The second group contains three benchmark programs: ANT, ATM and DNS from literature [2].

For the first group, we implement three kinds of cycle-breaking strategies as competitors. For all dependencies that are involved in cycles:

- NC removes the dependency involved in the greatest number of cycles.
- SCplx deletes the dependency with the minimal stubbing complexity.
- CWR proposed by Abdurazik et al. [11] removes the dependency with maximal cycles-weight ratio.

If multiple candidate dependencies exist, three strategies randomly choose one to remove.

For the second group, HBCITO is compared to two typical class integration test order generation methods:

- GA proposed by Briand et al. [2] applied genetic algorithm to minimize the overall stubbing complexity of the class test orders.
- A graph-based method proposed by Abdurazik et al. [11] eliminated cycles by removing the edge with the highest cycles-weight ratio, in order to break the most number of cycles with the minimal stubbing effort.

The overall stubbing complexity (OCplx) and number of required stubs (Stubs) are used to evaluate the stubbing cost of constructing test stubs based on generated class test orders. The lower the values of two indicators, the better the performance of the class integration test order generation approaches.

B. Results

Table II presents the range and mean values of OCplx and Stubs for three large-scale programs. Each method is executed 30 times for each program. The minimal mean values of OCplx and Stubs for each program have been highlighted in bold.

As shown in Table II, the mean values of OCplx obtained by HBCITO are the minimal for all three programs. CWR clearly outperforms NC and SCplx in terms of OCplx for all programs when it considers both the objectives of reducing total number of test stubs and minimizing stubbing cost for each removed dependency. However, calculation of cycles-weight ratio is a rough means to balance two objectives so that it is likely to miss a better result. On the other hand, the values of OCplx obtained by CWR and HBCITO are relatively deterministic compared with other two strategies, due to fewer arbitrary decisions in the process of cycle-breaking. For example, both of them obtain only one value of OCplx for programs JHotDraw and log4j3.

For program jmeter, it can be observed that not all solutions obtained by HBCITO perform better than that of

[1] http://sir.unl.edu/portal/index.html

Table II
COMPARISON OF RESULTS FOR LARGE-SCALE PROGRAMS

Systems	Methods	OCplx		Stubs	
		Range	Mean	Range	Mean
JHotDraw	NC	[1.580-2.186]	1.901	90	**90**
	SCplx	[0.977-0.997]	0.986	[94-97]	95.400
	CWR	0.963	0.963	[91-92]	91.700
	HBCITO	0.957	**0.957**	91	91
jmeter	NC	[1.857-2.665]	2.335	39	**39**
	SCplx	[1.467-1.588]	1.528	[55-62]	58.500
	CWR	[1.305-1.335]	1.320	[42-43]	42.500
	HBCITO	[1.284-1.325]	**1.304**	[40-44]	41.933
log4j3	NC	[2.440-3.036]	2.725	90	**90**
	SCplx	[2.140-2.403]	2.255	[102-118]	109
	CWR	1.972	1.972	96	96
	HBCITO	1.966	**1.966**	91	91

CWR, but the value ranges for HBCITO are smaller than that of CWR. Moreover, for the best result, HBCITO obtains 0.021 (1.305-1.284) lower OCplx than CWR.

In terms of Stubs, it is no doubt that NC would obtain the minimal number of test stubs. Except for NC, HBCITO constructed fewer test stubs compared with other two strategies, which means that HBCITO tries to balance the two objectives (the number of created test stubs and the corresponding stubbing complexity) when breaking cycles.

Table III
COMPARISON OF RESULTS FOR BENCHMARK PROGRAMS

Systems	Methods	OCplx		Stubs	
		Range	Mean	Range	Mean
ANT	GA	[3.59-3.86]	3.62	[12-14]	**12.16**
	Abdurazik	4.19	4.19	17	17
	HBCITO	[2.01-2.57]	**2.23**	[14-19]	16.32
ATM	GA	2.68	2.68	7	7
	Abdurazik	2.68	2.68	7	7
	HBCITO	[2.17-2.68]	**2.56**	[6-7]	**6.77**
DNS	GA	1.47	1.47	[6-7]	6.89
	Abdurazik	1.33	1.33	5	**5**
	HBCITO	[1-1.18]	**1.14**	5	**5**

Table III shows the comparison of results for three benchmark programs. The values of two quality indicators for GA are obtained from reference [2]. For other two approaches, each method is executed 100 times for each program to be consistent with experimental settings in [2]. The best results have been highlighted in bold.

In terms of OCplx, The mean values obtained by HBCITO are minimal for all programs compared with GA and Abdurazik et al.'s approach. All values of OCplx in HBCITO are lower than the best results of other two methods, although the range of OCplx in HBCITO is more varying. The reductions on OCplx achieved by HBCITO compared to the best results of the other two methods range from 0.12 (2.68-2.56) on program ATM to 1.36 (3.59-2.23) on program ANT.

In terms of Stubs, HBCITO obtains the minimal values on two programs. For program ANT, although the best solution obtained by HBCITO constructed two more stubs than that of GA, HBCITO achieves a lower overall stubbing cost since its test stubs emulating fewer attributes and methods.

Among six programs, the highest reduction on OCplx achieved by HBCITO compared to the best results of others is about relative 37.88% (1-2.23/3.59) on program ANT, while the lowest reduction is about 0.30% (1-1.966/1.972) on program log4j3. The average reduction on stubbing cost is 9.80% with the standard deviation as 0.15. It can be concluded that HBCITO performs better than competitors.

Table IV
RUNNING TIME OF HBCITO

Systems	Analysis (ms)	Identifying (ms)	Generation (ms)
JHotDraw	9743	119	484
jmeter	6346	659	731
log4j3	4467	1594	2106
ANT	-	66	429
ATM	-	11	54
DNS	-	14	50

Table IV shows the running time of HBCITO, including three parts: analyzing the program to obtain class dependencies and coupling information, identifying all cycles, and generating class test orders. For three benchmark programs, information about class dependencies was taken from the Appendix in literature [2], so the analysis time is null. All experiments are conducted on Intel AI DevCloud.

The analysis time of three large-scale programs is positively correlated with number of classes, since all class dependencies are analyzed. For identifying time, it is obvious that the more cycles, the more time spent. For ANT and jmeter, their number of cycles is similar, but jmeter has 309 SCCs while ANT has only 14 SCCs. Since HBCITO handles SCC one by one, the identify time of jmeter is more than that of ANT. For generation time, it is related to the number of cycles. But for JHotDraw, it takes a little long time, that is because it has more SCCs containing cycles.

C. Threats to Validity

Although the experimental results demonstrate the effectiveness of HBCITO, it still faces the following threats to validity:

Firstly, some parameters are used in the heuristic algorithm, such as the number of iterations in the exploration. The details about it are not shown due to limited space. The performance of HBCITO can be varied if different parameter values are used. Our next research direction is to verify the effect of different values of parameters on the performance of HBCITO.

Secondly, HBCITO is non-determinism in some cases as the results shown in Table II and III. That's because our algorithm can choose different edges as alternative that leading to various consequences. Therefore, if a software tester runs these algorithms once, then the output may not be the best solution. However, the experimental results show that even the worst solution generated by HBCITO, is still better than others in most cases. HBCITO might not be the best choice, but it is a good choice compared with other approaches.

Thirdly, HBCITO is evaluated on three large-scale programs and three benchmark programs, but we are unable to make sure the conclusions can hold for all programs. In the future, we intend to estimate our approach on more programs.

V. Conclusion

This paper considers the merits in existing search-based and graph-based approaches, and proposes a heuristic-based method called HBCITO to break cycles for the class integration test order generation. Our method first initializes a class dependency to remove a part of cycles, then heuristic algorithm is applied to search for a better alternative that breaks the same cycles or the same number of cycles as the initialized dependency, but attracts least stubbing cost. The search space is significantly minimized for complex programs containing hundreds of classes given HBCITO only considers certain dependencies that are involved in cycles. The experimental results show that HBCITO is able to obtain the class integration test orders with the minimal stubbing cost among all solutions generated by existing approaches.

Acknowledgment

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No.11208017) and the research funds of City University of Hong Kong (9678149, 7005028, and 9678149), and the Research Support Fund by Intel (9220097).

References

[1] H. Melton and E. Tempero, "An empirical study of cycles among classes in Java," Empir Software Eng, vol. 12, no. 4, pp. 389-415, 2007.

[2] L. C. Briand, J. Feng, and Y. Labiche, "Experimenting with genetic algorithms and coupling measures to devise optimal integration test orders," Tech. Rep. TR SCE-02-03, Carleton University, 2002.

[3] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima, "Class firewall, test order, and regression testing of object-oriented programs," J. Object-Oriented. Program, vol. 8, no. 2, pp. 51-56,1993.

[4] K. C. Tai and F. J. Daniels, "Test order for inter-class integration testing of object-oriented software," In Proceedings of the 21st International Conference on Computer Software and Applications. IEEE, 1997, pp. 602-607.

[5] Y. L. Traon, T. Jéron, J.-M. Jézéquel and P. Morel, "Efficient object-oriented integration and regression testing," IEEE Trans. Reliab, vol. 49, no. 1, pp. 12-25, 2000.

[6] L. C. Briand, Y. Labiche and Y. Wang, "An investigation of graph-based class integration test order strategies," IEEE Trans. Software Eng, vol. 29, no. 7, pp. 594-607, 2003.

[7] R. Hewett and P. Kijsanayothin, "Automated test order generation for software component integration testing," In Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering. IEEE, 2009, pp. 211-220.

[8] V. L. Hanh, K. Akif, Y. L. Traon, J.-M. Jézéquel, "Selecting an efficient OO integration testing strategy: an experimental comparison of actual strategies," In Proceedings of the 15th European Conference on Object-Oriented Programming. Springer-Verlag,2001,pp. 381-401.

[9] B. A. Malloy, P. J. Clarke, and E. L. Lloyd, "A parameterized cost model to order classes for class-based testing of C++ applications,". In Proceedings of the 14th International Symposium on Software Reliability Engineering. IEEE, 2003, pp. 353-364.

[10] P. Bansal, S. Sabharwal and P. Sidhu, "An investigation of strategies for finding test order during integration testing of object oriented applications," In Proceeding of International Conference on Methods and Models in Computer Science. IEEE, 2009, pp. 1-8.

[11] A. Abdurazik and J. Offutt, "Using coupling-based weights for the class integration and test order problem," Comput. J, vol. 52, no. 5, pp. 557-570, 2009.

[12] L. Borner and B. Paech, "Integration test order strategies to consider test focus and simulation effort," In Proceedings of the 1st International Conference on Advances in System Testing and Validation Lifecycle. IEEE, 2009, pp. 80-85.

[13] S. R. Vergilio, A. Pozo, J. C. Árias, R. V. Cabral, T. Nobre, "Multi-objective optimization algorithms applied to the class integration and test order problem," Int. J. Software Tools Technol. Trans, vol. 14, no. 4, pp. 461-475, 2012.

[14] W. K. G. Assunção, T. E. Colanzi, S. R. Vergilio, A. Pozo, "A multi-objective optimization approach for the integration and test order problem," Information Sciences, vol. 267, no. 2, pp. 119-139, 2014.

[15] G. Guizzo, G. M. Fritsche, S. R. Vergilio, A. T. R. Pozo, "A hyper-heuristic for the multi-objective integration and test order problem," In Proceedings of the 2015 International Conference on Genetic and Evolutionary Computation. ACM, 2015, pp. 1343-1350.

[16] T. Mariani, G. Guizzo, S. R. Vergilio, A. T. R. Pozo, "Grammatical evolution for the multi-objective integration and test Order problem," In Proceedings of the 2016 International Conference on Genetic and Evolutionary Computation. ACM, 2016, pp. 1069-1076.

[17] G. Czibula, I. G. Czibula and Z. Marian, "An effective approach for determining the class integration test order using reinforcement learning," Appl. Soft Comput, vol. 65, pp. 517-530, 2018.

[18] R. Tarjan, "Depth-first search and linear graph algorithms," In Proceedings of the 12th Annual Symposium on Switching and Automata Theory. IEEE, 1971, pp. 114-121.

[19] M. Črepinšek, S.-H. Liu, and M. Mernik, "Exploration and exploitation in evolutionary algorithms: A survey," ACM Comput. Surv., vol. 45, no. 3, pp. 1-33, 2013.