Hindawi
Scientific Programming
Volume 2022, Article ID 3708422, 11 pages
https://doi.org/10.1155/2022/3708422

Research Article

Multiobjective Genetic Algorithm for Class Testing using OCL Class Contract Specifications: A Framework

Syed Muhammad Saqlain Shah,[1] **Rehan Farooq**,[1] **Abdullah Alharbi**,[2] **Hashem Alyami**,[3] **Islam Zada**,[4] **and Faiz Ali Shah**[5]

[1]*International Islamic University, Islamabad, Pakistan*
[2]*Department of Information Technology, College of Computers and Information Technology, Taif University, P.O. Box 11099, Taif 21944, Saudi Arabia*
[3]*Department of Computer Science, College of Computers and Information Technology, Taif University, P.O. Box 11099, Taif 21944, Saudi Arabia*
[4]*Department of Computer Science, University of Peshawar, Peshawar 25120, Pakistan*
[5]*University of Tartu, Tartu, Estonia*

Correspondence should be addressed to Syed Muhammad Saqlain Shah; syed.saqlain@iiu.edu.pk

Received 27 December 2021; Accepted 16 February 2022; Published 26 April 2022

Academic Editor: Zhongguo Yang

It has been a software trend to build large-scale complex systems with high reliability. Due to the size of the software and the dynamic requirements of the stakeholders, it becomes hard to test those software systems manually. This may lead the software to fatal failures and cause irrecoverable catastrophic damage. To be safe, the software system must be investigated thoroughly before it is too late. Test sequence generation for Unified Modeling Language (UML) class models from their semiformal Object Constraint Language specifications can be helpful in identifying the defects in the early phase of the software life cycle. The existing approaches suffer from inherent problems of exhaustive exploration of finite state machines (infeasible paths, exponential number of test sequences, and uncertainty of completion of testing). Evolutionary algorithms can greatly help by optimizing the test sequences to get optimal coverage, minimal cost, and higher quality. The proposed approach helps us to improve the testing of Unified Modeling Language (UML) model-based software, by testing the conformance to semiformal class operation contract specifications (specified in the form of Object Management Group (OMG) standard and Object Constraint Language (OCL) semiformal language). The presented research achieved two main goals: (1) automation of testing process and conformance to standards of the current technique of test sequence generation, bridging the gap between the research and industry; (2) improvement in the state of the art approach through the application of multiobjective genetic algorithms (MOGAs). A case study along with the results achieved through the proposed technique is presented as well, clearly reflecting the significance of the proposed research.

1. Introduction

Specification-based testing refers to the area of software testing where software is tested against its specification. It is functional black-box testing where software is tested on its interfaces to validate the documented requirement specifications. It deals with generating test suites from the software specifications, executing the test case scenarios against the actual software, and then checking the results against test oracles. Specification-based testing takes advantage of building the testing environment of software prior to its existence [1–4]. In model-based testing, software is represented in terms of models. Unified Model Language (UML) is one of the known modeling techniques which models the software in static (e.g., class diagrams) and dynamic (e.g., sequence diagrams) structures. Model-based development lets the engineers focus on domain specific issues comparable to the technical issues of software development. The

support of existing tools makes model-based development and model-based testing more attractive. The available tools help the engineers to model the software, transform software models from one representation to another, and generate abstract test cases from the software model. These abstract test cases can then be transformed into actual executable tests [2, 5]. Traditionally, GAs have been used as a search heuristic in finding the optimal set of solutions for single objective problems. Recent advances in the field suggest the usage of GAs for multiobjective optimization [5]. In principle, multiobjective GAs (MOGA) are the same as GA based tools, but the potential solutions are evaluated for multiple parameters, and their fitness values are evaluated by multiple fitness functions. The MOGA evolution process involves the comparison of multiple fitness values for candidate chromosomes. Multiobjective optimization is particularly useful in the problems where the optimization of objective is not possible without compromising the quality of competitive objective(s). The generated solutions are referred to as Pareto optimal solutions. Test sequence optimization involves trade-off between testing cost and achieved test coverage, so the process is a strong candidate for multiobjective optimization [6]. In this paper, the proposed framework focuses on the following objectives: (1) to improve the unit testing of class models using OCL class contract specifications in compliance with industry standards, (2) to automate the test sequence generation process, (3) to reduce the states from infeasible test sequences, and (4) to improve the test coverage [7, 8].

The presented research achieved two main goals: (1) automation of testing process and conformance to standards of the current technique of test sequence generation, bridging the gap between the research and industry; (2) improvement in the state of the art approach through the application of multiobjective genetic algorithms (MOGAs). A case study along with the results achieved through the proposed technique is presented as well, giving clear reflection of the significance of the proposed research.

The rest of this paper has following organizational structure: Section 2 presents a review of existing techniques; Section 3 is all about the proposed framework; Section 4 presents experiment, results, and discussion; and Section 5 concludes the research.

2. Literature Review

In this section, a review of existing methodologies is presented. The review is split into two specializations, i.e., test sequence generation and test sequence optimization.

2.1. Test Sequence Generation. Generation of test sequences/test cases is among the challenging tasks for a test engineer. It involves trade-offs between the number of test cases and the desired test coverage, available resources, quality of test cases, achieved coverage, etc. Zhao et al. [9] aimed to develop the infrastructure of automatic test data generation for extended finite state machine (EFSM) models that produce real data to trigger feasible transition paths. It provided empirical results on efficiency analysis of test data generation for a set of state-based models. Derderian et al. [10] proposed automated unique input output (UIO) sequence generation for finite state models. The sequence generation problem was regarded as a search problem and was targeted through genetic algorithms (GA). The authors used 11 real and 23 randomly generated FSMs as proof of concept experiments. The GA based experimental results showed 62% better performance compared to the random search.

UML diagrams model both the static and dynamic aspects of a system. Asthana et al. [11] proposed a novel technique to generate the test cases from class and sequence diagrams. The research emphasized that the input diagrams were not transformed into some intermediate model which seems to be compromised as the model was represented in XML form which seems to be an intermediate form. Mingsong et al. [12] generated test cases from UML activity diagrams. They presented a comparison between the dynamic behavior of the activity diagram and actual program execution. Shah et al. [13] worked on automatic testing. They proposed a framework for extracting the test cases from UML diagrams of sequence and class hierarchy. They exported UML diagrams to XML format; then, from those XML files, use cases were generated. Kumar et al. [14] proposed a methodology which generated test cases by using class, sequence, and activity diagrams. They generated XML structure of diagrams defined through the UML. Three types of test cases were generated, i.e., static, dynamic, and integrity test cases. Static test cases were generated from class diagrams, while activity and sequence diagrams were used for integrity test cases and dynamic test cases. Hilken et al. [15] have argued the importance of modeling languages, e.g., OCL and UML, and their role in designing a system. They pointed out that the modeling languages provide a lot of description through a large number of constructs. Gupta [1] proposed an approach to test the class methods interaction through class contracts. The author used a state-based approach. An abstract state configuration of class and an initial abstract state were used to incrementally generate the reachable states. The task was accomplished through the search of the methods which can be invoked in the current state. It resulted in inability to match the automation and syntax of OCL standards, and the standard parsers were unable to parse the syntax. The approach used AFS traversal to generate test sequence paths and hence faced inherent problems of finite state. The author has used the traditional searching approach for path traversal of finite state machines, and all-transition coverage has been used as sequence path generation. A specification-based testing approach was proposed, which used class contracts specified in the form of OCL constraints (class invariants, preconditions, and postconditions). Miller and Strooper [2] presented a case study on specification-based implementation testing framework. The authors argued that the proposed framework produced almost the same performance as the resulting BZ-Testing tools. They emphasized that their framework was more cost effective than the manual testing. Ali et al. [16] worked on model-based testing (MBT) by generating the test data through OCL constraints. Their

focus was to work in the industrial domain through applying the heuristic search for generating test data and automating MBT. The accomplished tasks were evaluated through three algorithms, i.e., (1 + 1) evolutionary algorithm, genetic algorithms, and alternating variable method. Dang and Gogolla [17] presented a framework using OCL for model transformation. At the baseline, it integrated OCL and Triple Graph Grammars. They have used it for verification and quality assurance of model transformation.

2.2. Test Sequence Optimization. Di Geronimo et al. [18] emphasized that the use of search-based software testing becomes highly computational overhead when applied over large applications. To solve the computational overhead issue, specifically in cloud environments, the authors presented parallel GA. The solution was presented through Hadoop MapReduce. Harman et al. [4] proposed three search-based algorithms for test data generation and presented the results of a case study in support of their approach. The authors emphasized that their approach maximized the coverage and minimized the required number of test cases. The size of the software considered for case studies was as big as 144 lines of code, which might be good for a proof of concept. Arcuri et al. [19] focused on comparison of three test automation strategies, namely, random testing, adaptive random testing, and search-based testing using genetic algorithms. The results of the abovesaid strategies were presented as well. Srivastava [20] presented a GA based approach for test data generation which took the user input variables and generated the test data. The author claimed that GA outperforms random testing on time measures. Srivastava et al. [21] presented another search-based test sequence generation technique using the ant colony optimization algorithm. The "ants" were used to explore CFG and find the optimized test sequences. Yano et al. [22] presented an approach of test sequence generation using evolutionary algorithms. They presented an evolutionary approach for test sequence generation from a behavioral model, in particular, EFSM. A multiobjective evolutionary algorithm, M-GEO vsl, adopted from M-GEO, was used; it considered two objectives: to search for a test sequence that covers a target transition and to minimize the length of this test sequence [23]. Literature reveals that UML diagrams are not sufficient enough to specify complete class behavior; most accurate details of a class are revealed from the OCL class specifications in the form of OCL class contracts [1].

3. Materials and Methods

The proposed approach uses the OMG's (Object Management Group) standard OCL syntax and automation standard of the test sequence generation. In order to improve testing effectiveness, this paper applied a multiobjective approach using MOGA where optimization of the test coverage and validity of the test sequences constitute a concern. Our goal is to produce test sequences which are most effective in identifying and revealing software implementation problems. The proposed approach is divided into two main phases as shown in Figure 1; i.e., (1) standard OCL parsing is done on the input OCL class contracts, and an abstract finite state machine is generated; (2) state-based test sequences, generated from the source AFSM using multiobjective GA, are optimized.

3.1. Parsing Class Contracts and Generating Abstract Finite State Machine. The proposed technique takes class contracts as input, and by using the standard OCL parser [24], a parse tree of the input class contracts is generated. The parser is used with Eclipse [25], IDE for Java, for parsing the OCL constraints of UML models. The input OCL class contracts are in textual form. The generated parse tree is subject to its semantic analysis and construction of domain specific objects. A parse tree processor is implemented using Java that transverses the parse tree and extracts the objects corresponding to the domain concepts of OCL semantics. Figure 2 shows the class diagram of mapping objects of OCL operation contracts. The objects shown are extracted through the implemented processor. After generation of parse tree, there is the process of semantic analysis of the output parse tree and construction of domain specific objects in Java. The proposed OCL parse tree processor transverses the parse tree and extracts the objects corresponding to the domain concepts of OCL semantics. Next to the generation of parse tree, the abstract finite state machine is constructed by applying the rules from [1].

The abstract state model of the software from specification is created by starting from the class constructors. For each constructor, a new initial state in the abstract finite state machine is created, and then all states resulting from initial state onward are dynamically created. The proposed framework deviates from the existing research [1] that suggested transition tree coverage criterion; i.e., test sequences are identified along with the simple paths. Simple path coverage misses the self-reference transitions, and it is quite possible that a method might fail on subsequent invocations as the subsequent calls may bring the object in as state (due to implementation faults) that it might behave anomalously; even the specifications may suggest some other behavior. However, in case there are self-transitions to a state, it might skip a valid step in the sequence of method calls. Therefore, it is better if there are row test sequences from exhaustive search of the AFSM. The test sequences generated in this step are used as an initial population for the MOGA optimization [26].

3.2. Coding Test Sequences in Chromosomes and Optimization through MOGA. After buildup of the abstract finite state machine, the next phase is generation and optimization of the testing sequences. This phase involves coding the test sequences in tool specific chromosomes, executing MOGA, and selecting the best fit test sequences after evolution.

3.2.1. Coding Solutions in Genes and Chromosomes. The proposed approach devised a coding scheme where a

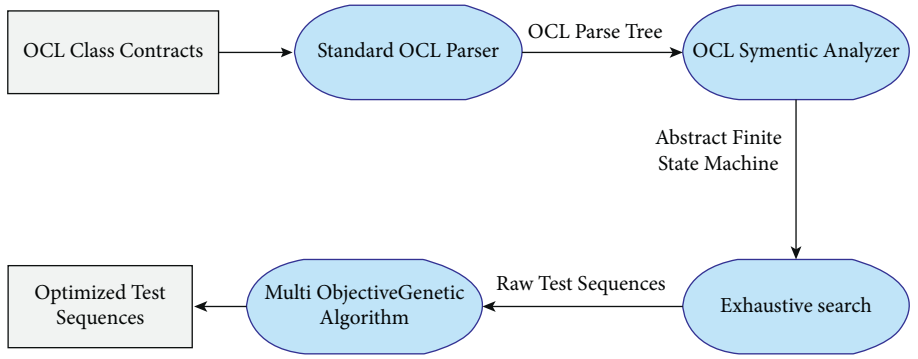

Figure 1: Proposed automated MOGA optimized test sequence generation process.

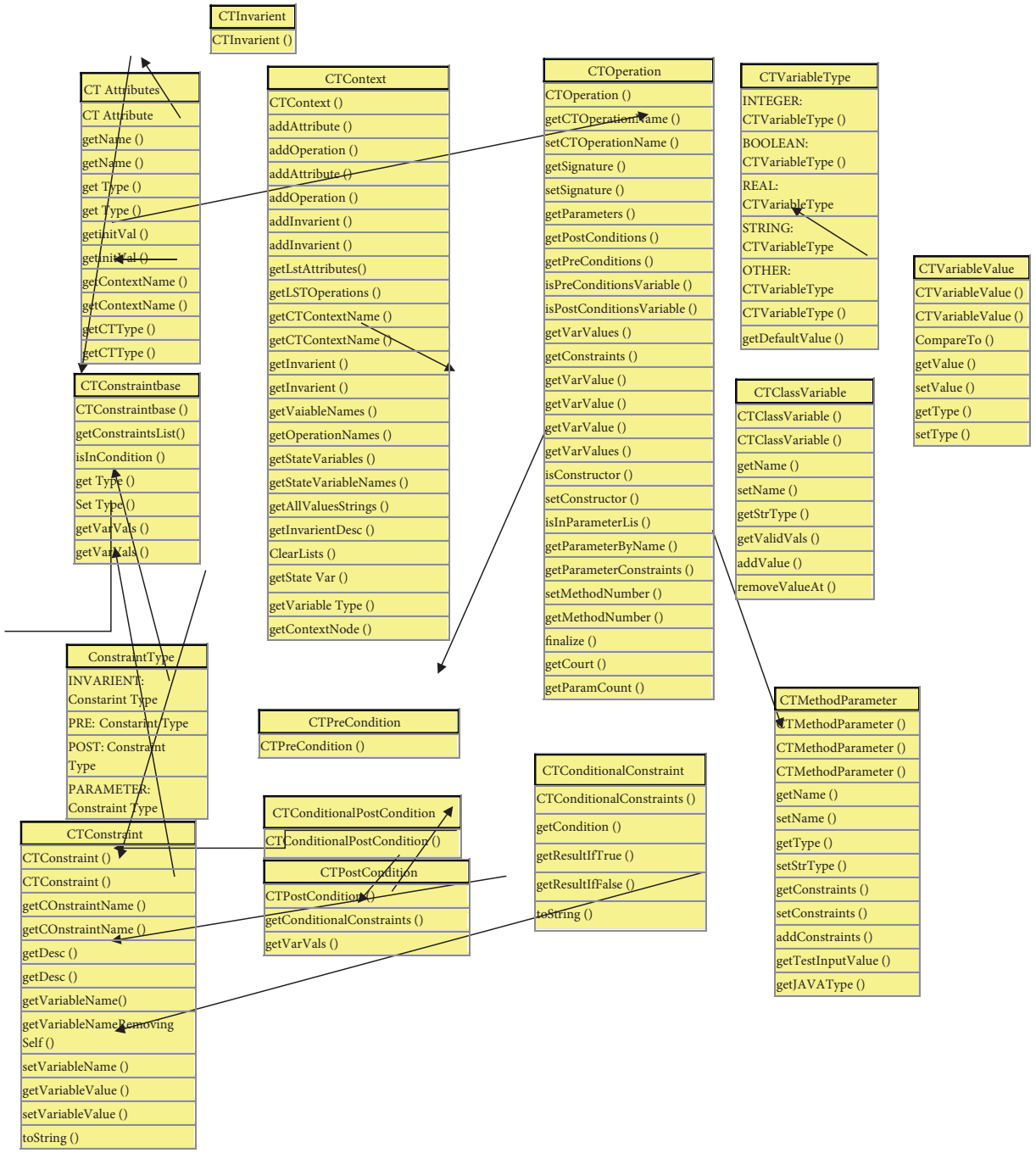

Figure 2: The class diagram of mapping objects of OCL operation contracts.

potential solution (chromosome) comprises transition (test transition) from the built abstract finite state machine. Each test transition represents a transition in the abstract finite state machine with the method represented by this transition.

We have used the JGAP tool to evolve the population having randomly generated chromosomes. It generates the chromosomes of length n, where each of the genes is represented by test transition objects. At first, a random state is picked out of all the states of the finite machine generated in the last step. Next, gene is one of the outgoing edges from the selected state, and this transition is again chosen randomly. This process goes on till all the n genes are coded. A potential chromosome in our solution set can be visualized as in Figure 3.

Each transition, T_i, in the coding scheme contains reference to the initiating state (transition from state) from which that transition originated and a reference to the terminating state to which that transition is leading, where n is the length of the chromosome, T_i is the ith transition in the test sequence, and $i = 1,2,3, \ldots, n$.

T1	T2	T3	⋯	Tn

Figure 3: A chromosome of length n in our coding scheme.

3.2.2. The Multiple Objectives. In order to get quality test sequences, the proposed approach has two objectives which do not conflict, but optimization of one may decrease the fitness of the other objective. The objectives of the proposed technique are as follows:

(1) Coverage Optimization. While testing, the proposed approach is interested in revealing all possible errors by applying all possible input combinations to the method interface of the class under test (CUT). Due to infinitely many combinations of class state variables and method input parameter values, it is practically impossible to test all possibilities. The proposed approach can only have as improved class test coverage as possible so that the level of the quality of our testing process is ensured. Therefore, the first of the two objectives is the optimization of generated test sequences in terms of the coverage. Our fitness function evaluates the number of transitions of the finite state machine covered by the test sequence.

$$\text{Coverge fitness (CF)} = \sum_{i=1}^{n} (\text{coverage weight for call sequence}). \tag{1}$$

In order to calculate coverage fitness, coverage weights are used which may be added according to three different scenarios:

(i) If a transition is covered once, chromosome is given additional positive weight-age, and it rewards a chromosome for covering a transition.

(ii) If a transition is not covered at all by a chromosome, it is given additional negative weight-age, and it rewards a chromosome negatively for not covering a transition.

(iii) If a transition is covered more than twice by a chromosome, it is given additional negative weight-age, and it rewards negatively due to repetition.

(2) Test Sequence Order Optimization. Comprehensive testing of a class involves testing for both valid and invalid method interactions [1]. By inherent properties, MOGA searches through the solution space by building random solutions based on the genetic operators. In the case of class unit testing, any sequence of method calls may be valid, but a question arises as to getting test sequences which are in sequence according to their place in the finite state model. Our second objective is to make the test sequences as in order as possible. Fitness value of solution by assessing its order often is in contrast with the fitness value for overall coverage achievable by that solution.

$$\text{Oreder fitness (OF)} = \text{initial state weight} + \sum_{i=1}^{n} (\text{coverage weight for call sequence}). \tag{2}$$

Description of the weight calculation for test sequence order fitness is given as follows:

(i) Initial state weight: if the first gene of the chromosome has an initial state of AFSM as from state, then this weight is added; otherwise, it is skipped.

(ii) Sequence weight for call sequence: we calculate the quality of chromosome by the sequence of method calls and reward each chromosome by the following formula:

(a) If any of the method calls (genes) is in a valid sequence, then a positive weight is added to the second fitness value.

(b) If any of the method calls (genes) is not in a valid sequence, then a negative weight is added.

3.3. The Genetic Evolutionary Process. The evolution process in our approach is completed by the following steps. This genetic evolution of chromosomes is done automatically by Java Genetic Algorithms Package (JGAP) [27].

3.3.1. Initialization of Test Sequence Population. Initial population of the test sequences can be generated either completely at random where transitions from the generated AFSM are picked at random to create genes of each chromosome of the initial chromosome pool. We have initialized the pool by exhaustive search of the AFSM. This initialization is used to minimize the possibility of evolution of the population toward local maxima.

3.3.2. Selection for Reproduction. Process of selection involves selection of fittest individuals for mating in the next population. Here, each gene is passed from the genetic evolution tool to our fitness function evaluator and is then assigned fitness values based on our fitness functions.

3.3.3. Reproduction of Population. Population created in step one undergoes genetic processes of crossover and mutation and gets evolved over generations. After each generation, chromosomes are assigned fitness values according the fitness functions.

3.3.4. Crossover. Based on the selected crossover probability, a single point crossover is performed on the population chromosome, where parts of the chromosomes are swapped and new offspring are created for next generation selection. Here we have used 0.30 as the crossover probability. The crossover process is shown in Figure 4.

3.3.5. Mutation. In this operation, value(s) of genes are mutated based on the mutation probability, and resulting chromosomes are constructed. In the proposed model, mutation probability is set to 0.07. Here, some of the test transition objects in the target chromosome are replaced with randomly selected values from the AFSM. Our random transition selection mechanism plugs in with the evolution tool and provides random transitions when required for mutation purpose. Example of random mutation is shown in Figure 5.

3.3.6. Termination Condition. The proposed approach uses the termination criterion of evolving the population, i.e., specific number of times, while reproducing the individuals.

3.4. The Fitness Functions. In order to optimize the test sequences through MOGA, the role of efficiently defined fitness functions is critical. As in the adopted MOGA, two objectives are to be achieved, so we have defined two fitness functions, i.e., (1) fitness by coverage and (2) fitness by test sequence order. The required fitness values are calculated through the two presented algorithms: coverage-based chromosome fitness calculation algorithm (CCFA) (Algorithm 1) and test sequence order-based chromosome fitness calculation algorithm (TSOCFA) (Algorithm 2), which is already used for the same problem by [28, 29].

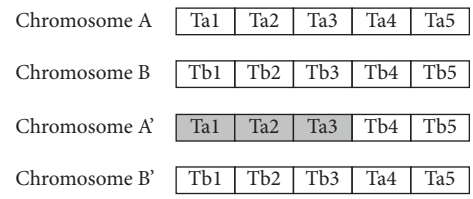

Figure 4: Sample crossover process.

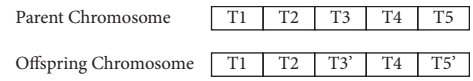

Figure 5: Sample mutation process.

4. Case Study and Experiment

Generation of test sequences is a critical part of the testing phase of software development life cycle. The test sequences for unit testing of a class can be generated from OCL class specifications [1], i.e., by mapping class specifications (OCL class contracts) to the class model (specifically a class in the class diagram). Existing test sequence generation process [1], when applied to actual testing, reveals some critical issues. Those issues and tackling them through the proposed framework are presented in the form of a case study. The CoinBox class is taken from a Drink Vending Machine's class diagram; this class is responsible for keeping record of the number of available drinks and quarters entered by the customer. The reason to use CoinBox class lies in presenting a fair comparison with existing research [1] where it is used for approach verification. Two more classes, Stack and Circle, were tested as well.

In the following code of block, the represented syntax is used by [1] and deviates from the standard OCL [2] in many aspects like the following: (1) Each statement in pre- and postconditions must be joined by a logical operator, e.g., "and", which is missing in the example. (2) Standard OCL syntax does not allow the use of curly braces "{}" around the context declarations. (3) All the OCL contexts (equivalent to class) must be declared inside a package and endpackage statement. (4) Each constraint in the invariant declaration must be separated by "and" instead of ",". (5) Writing just ":" operator while declaring a method signature is not enough; it should be fully qualified with the context name being referred to by the method. (6) Each "if" must have an accompanying "else" in order to validate OCL statement.

Context CoinBox{

Int curQtr, quantity, totalQtrsboolean allowVend Inv.int curQtr, quantity, totalQtrs ≥ 0

::CoinBox()

Post: self.curQtr = 0

```
INPUT:
One State Coverage Weight (wCoveredOnce),
Weight for state covered Twice (wCoveredTwice)
Weight for state covered more than Twice (wCoveredMoreThanTwice)
OUTPUT: Chromosome Fitness (CF)
(1) Initialize CF = 0
(2) for each Chromosome c in the current population do
(3)   for each Gene g in c do
(4)     if g occurs once then
(5)       CF = CF + wCoveredOnce
(6)     end if
(7)     if g occurs twice then
(8)       CF = CF + wCoveredTwice
(9)     end if
(10)    if g occurs more than twice then
(11)      CF = CF + wCoveredMoreThanTwice
(12)    end if
(13)  end for
(14)  Set coverage fitness of c equals CF
(15) end for
```

Algorithm 1: Chromosome fitness value by coverage (CCFA) [30].

```
INPUT: Weight for State in Start (wSState),
Weight for State in Sequence (wInSeq)
Weight for state not in Sequence (wNotInSeq)
OUTPUT: Chromosome Fitness (CF)
(1) Initialize OF = 0
(2) for each Chromosome c in the current population do
(3)   if c starts with an initial state then
(4)     OF = OF + wSState
(5)   end if
(6)   for each Gene g in c do
(7)     if g is in sequence then
(8)       OF = OF + wInSeq
(9)     else
(10)      OF = OF + wNotInSeq
(11)    end if
(12)  end for
(13) Set order fitness of c equals OF
(14) end for
```

Algorithm 2: Chromosome fitness value by test sequence order (TSOCFA) [31].

Self.allowVend = FALSE
Self quantity = 0
Self.totalQtrs = 0
::addQtr():void//add a quarter in the machine
Pre :self quantity >0;
Post :self.curQtr = curQtr@pre +1
If(self.curqtr@pre = 1) then
Self.allowVend = TRUE)
::retRtrs():void//return quarters back to the user
Pre :self.curQtr >0;
Post :self.curQtr = 0
Self.allowVend = FALSE
::vend():void//deliver a drink
Pre :self.allowVend = = TRUE and
Self.quantity >0;
Post:self.curQtr = 0
Self.allowVend = FALSE
Self.quantity = quantity@pre − 1
Self.totalQtrs = totalQtrs@pre + curQtrs@pre
::addDrink(m:int):void//add *m* users of drink in the machine
Pre :self.quantity = = 0 and $m > 0$
Post :self.quantity = quantity@pre + m
}

As already mentioned in [1] that the above block of code was not fulfilling the OCL standard syntax, the proposed framework modified this code syntax as adopting all the requirements of the standard OCL syntax. The resulting OCL class contract is acceptable according to the OCL 2.0 standard. The proposed approach used mutation analysis for benchmarking its performance. Moreover, fault seeding in classes under the test is done through Mu Java. It is worth noting that proper selection of the number of generations is problem specific and is important; e.g., a test run of the tool over CoinBox class gave 2 unique test sequences over 100 evolutions, but they got improved and diverse with 500 and 1000 generations.

Package CB

Context CoinBox

Inv: curQtr ≥ 0 and quantity ≥ 0 and totalQtrs ≥ 0

Context COinBox::CoinBox()

Post: Self.curQtr–0 and self.allowVend–FALSE and self.quantity–0 and self.total Qtrs–0

Context CoinBox::addQtr():void

Pre: Self.quantity > 0

Post: self.curQtr = curQtr@pre +1 and

If(self.curQtr@pre-1) then self.allowVend–TRUE else self.allowVend–FALSE end if

Context CoinBox::retQtrs():void

Pre: self.curQtr>0

Post: self.curQtr–0 and

Self.allowVend–FALSE

Context CoinBox: :Vend(): void

Pre: self.allowVend–TRUE and

Self.quantity >0

Post: self.curQtr–0 and

Self.allowVend–FALSE and Self.-quantity–quantity@pre–1 and Self.-totalQtrs–totalQtrs@pre + curQtr@pre

Context CoinBox: :addDrink (m: int): void Pre: self.-quantity = 0 and $m > 0$ Post: self.quantity–quantity@pre + m

endpackage

4.1. Results and Discussion. In this analysis, a predefined number of faults were seeded in the compiled class files. These faults were based on predefined mutation operators. From Table 1, the results reveal that the proposed approach attains better results in identifying the seeded faults. The reason behind the low performance of the existing technique [1] lies in its transition tree coverage which skips loops in the AFSM. Syntax of OCL used by existing approach [1] fails to be accepted as standard OCL syntax and fails to get parsed by the available OCL parsers. It deviates from the standard of writing OCL statements and hence cannot be employed in practical test sequence generation scenarios. The very first impact of that nonstandard OCL reveals the syntax errors. The proposed tool reads standard OCL constructs and automatically generates the test sequences applying the rules used by the current approach. It also allows on-demand optimization of the test sequences if desired by the test engineer. An obvious advantage of the automation along with effort saved from manual works is automatic changes to the test sequences on change of OCL specifications.

The experimental case study, exhaustive state space search, generated 872 test sequences with a maximum length of 26 with redundant test sequence loops. Much effort needs to be spent on executing all these test sequences. Application of MOGA with a population size of 25 and a sequence length of 15 gave 25 test sequences with a length of 15, each being optimized for all state coverage and ordered sequence paths over 1000 MOGA generations. It was also observed that more generations give more diverse test sequences with higher fault revealing efficiency. Since the proposed approach used a random population out of the search-based sequences, it minimizes the chances of bad genes and evolution in negative direction. A mutation analysis of the class under test found that MOGA based test sequences seem to give at least comparative defect revealing efficiency and may considerably outperform test sequences generated from the current approach. It is important to be noted that proper selection of number of generations is important; more generations might give better results but with considerable MOGA execution time.

By nature, as all optimization techniques, it is never expected that one gets an exact solution, but an optimized solution is obtained. MOGAs, being a subset of evolutionary algorithms, start with a possible set of solutions and try to optimize the set of solutions generation after generation. Evolution as mimicry of the natural process of evolution might not find suitable chromosomes (e.g., due to mutation) and might give some useless test sequences; this can be controlled using better fitness functions. This is obvious because in nature if wrong genes get to the next generations, then the individuals may suffer from defects. After generation of AFSM, it can

(i) either generate a stochastic random population where each chromosome is composed of a completely random set of genes.

(ii) or get a random population out of the population of test sequences generated from state-based test sequence generation approach.

The second option seems to give better results. While specifying MOGA fitness functions for test sequence optimization, the sequence of genes while calculating fitness values must be taken into account. The proposed approach gives improvement in terms of automation of test sequence generation process. MOGAs are quite effective while being used for test sequence optimization process, but the

Table 1: Mutation analysis of CoinBox, Stack, and Circle classes.

Class under test	Total faults seeded	Faults identified by Gupta [1]	Faults identified by proposed approach
CoinBox	117	81	83
Stack	73	51	59
Circle	98	55	53

Table 2: Comparative analysis of proposed framework with existing research.

	Automation	Specification based	Coverage	State based	Optimization	Multiobjective
Gupta [1]	✓	✓	✓	✓	×	×
Yano et al. [22]	✓	×	✓	✓	✓	✓
Harman et al. [4]	×	×	×	✓	✓	×
Asthana et al. [11]	✓	×	×	✓	×	×
Mingsong et al. [12]	✓	×	✓	✓	×	×
Derderian et al. [10]	✓	×	×	✓	✓	×
Shah et al. [13]	✓	×	×	✓	×	×
Proposed framework	✓	✓	✓	✓	✓	✓

proposed approach recommends the use of raw test sequences as initial population. MOGA optimized test sequences give optimized coverage within limited test sequence length and numbers.

Table 2 presents the comparative analysis of the proposed framework with the existing techniques in terms of six features, i.e., automation, being specification based, being coverage based, being state based, optimization, and being multiobjective. It may clearly be observed that all the techniques are either fully or partially automated except that presented by Harman et al. [4]. The proposed technique generates the test sequences in an automated mechanism. Next, only two of the techniques use OCL class specification for the test case generation, i.e., the proposed technique and research presented by Gupta [1]. UML diagrams were used in [22], class and sequence diagrams were used for the generation of sequences of test cases [11], activity diagrams were used by Mingsong et al. [12], and class and sequence diagrams were used in [13]. From the existing literature, it has already been established that the most accurate details of a class are revealed from the OCL class specifications in the form of OCL class contracts. As far as state coverage is concerned, only the proposed technique along with those in [1, 12, 22] takes state coverage into account while generating the test cases, and the rest of the techniques do not use this feature. As all the techniques along with the proposed technique either use FSMs or directed graphs during the process of test case generation, all of them use state-based features. The most optimized test cases are generated through [4, 10, 22] and the proposed framework while the rest of the comparative techniques use searching mechanisms which have their own inherent problems. The proposed technique and the research presented in [22] are multiobjective while all others are limited to single objectives. The multiobjective approach tries to limit the minimum number of states in test sequences by providing the maximum coverage. This comparison clearly reflects the fact that the proposed technique is better in terms of key features than its comparative ones.

5. Conclusion

The proposed approach has improved the existing approaches by conformance to industry standard syntax and automation from OCL to the actual test sequence generation. The proposed approach provides the advantage of optimization for test sequences in terms of minimum number and higher quality along with automation of test sequence generation process and conformance to industry-practiced OMG standard OCL syntax. It saves time and resources spent on the part of testing process where selection of test sequences is to be accomplished. Our approach gives improvement in terms of automation of test sequence generation process. Multiobjective genetic algorithms are quite effective while being used for test sequence optimization process, and use of raw test sequences as initial population appears to give better results compared to the completely random selection of initial population of test sequence chromosomes. MOGA test sequences give optimized coverage (maximum transition coverage) within limited test sequence length and numbers. The proposed framework can be used either by industry practitioner test engineers for creating test sequences while testing the software or by researchers while experimenting with FSMS, GA, and MOGAs.

This research can be improved using more reliable testing techniques for testing the software with FSMS, GA, and MOGAs.

Data Availability

No data are available.

Disclosure

The presented paper is a part of master's in software engineering thesis, submitted to the Department of CS & SE, International Islamic University, Islamabad, Pakistan [32].

Conflicts of Interest

The authors declare no conflicts of interest.

Acknowledgments

We deeply acknowledge Taif University for supporting this research through Taif University Researchers Supporting Project number (TURSP-2020/231), Taif University, Taif, Saudi Arabia.

References

[1] A. Gupta, "An approach for class testing from class contracts," *Automated Technology for Verification and Analysis*, vol. 6252, pp. 203–217, 2010.

[2] T. Miller and P. Strooper, "A case study in model-based testing of specifications and implementations," *Software Testing, Verification and Reliability*, vol. 22, no. 1, pp. 33–63, 2012.

[3] A. Konak, D. W. Coit, and A. E. Smith, "Multi-objective optimization using genetic algorithms: a tutorial," *Reliability Engineering & System Safety*, vol. 91, pp. 992–1007, 2006.

[4] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proceedings of the In Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on. IEEE*, pp. 182–191, Paris, France, April 2010.

[5] Y. Gao, L. Shi, and P. Yao, "Study on multi-objective genetic algorithm," *Proceedings of the 3rd World Congress on Intelligent Control and Automation*, vol. 1, pp. 646–650, 2000.

[6] J. A. Whittaker, "What is software testing? And why is it so hard?" *IEEE software*, vol. 17, no. 1, pp. 70–79, 2000.

[7] Z. Yin, J. Wu, J. Song, Y. Yang, X. Zhu, and J. Wu, "Multi-objective optimization-based reactive nitrogen transport modeling for the water-environment-agriculture nexus in a basin-scale coastal aquifer," *Water Research*, vol. 212, Article ID 118111, 2022.

[8] G. Yu, L. Ma, Y. Jin, W. Du, Q. Liu, and H. Zhang, "A survey on knee-oriented multi-objective evolutionary optimization," *IEEE Transactions on Evolutionary Computation*, p. 1, 2022.

[9] R. Zhao, M. Harman, and Z. Li, "Empirical study on the efficiency of search based test generation for efsm models," in *Proceedings of the Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pp. 222–231, Paris, France, April 2010.

[10] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo, "Automated unique input output sequence generation for conformance testing of fsms," *The Computer Journal*, vol. 49, no. 3, pp. 331–344, 2006.

[11] S. Asthana, S. Tripathi, and S. K. Singh, "A novel approach to generate test cases using class and sequence diagrams," in *Proceedings of the International Conference on Contemporary Computing*, pp. 155–167, Noida, India, August 2010.

[12] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic test case generation for uml activity diagrams," in *Proceedings of the 2006 International Workshop on Automation of Software Test*, pp. 2–8, Shanghai, China, May 2006.

[13] S. A. A. Shah, R. K. Shahzad, S. S. A. Bukhari, and M. Humayun, "Automated test case generation using uml class & sequence diagram," *British Journal of Applied Science & Technology*, vol. 15, no. 3, 2016.

[14] B. Kumar, K. Singh, B. Kumar, and K. Singh, "Testing uml designs using class, sequence and activity diagrams," *International Journal for Innovative Research in Science and Technology*, vol. 2, no. 3, pp. 71–81, 2015.

[15] F. Hilken, P. Niemann, M. Gogolla, and R. Wille, "From UML/OCL to base models: transformation concepts for generic validation and verification," in *International Conference on Theory and Practice of Model Transformations*, pp. 149–165, L'Aquila, Italy, July 2015.

[16] S. Ali, M. Zohaib Iqbal, A. Arcuri, and L. C. Briand, "Generating test data from ocl constraints with search techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1376–1402, 2013.

[17] D.-H. Dang and M. Gogolla, "An ocl-based framework for model transformations," *VNU Journal of Science: Computer Science and Communication Engineering*, vol. 32, no. 1, 2016.

[18] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, "A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites," in *Proceedings of the Software Testing, Veri_cation and Validation (ICST), 2012 IEEE Fifth International Conference on*, pp. 785–793, Montreal, QC, Canada, April 2012.

[19] A. Arcuri, M. Z. Iqbal, and L. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *Proceedings of the IFIP International Conference on Testing Software and Systems*, pp. 95–110, Natal, Brazil, April 2010.

[20] P. R. Srivastava, "Optimization of software testing using genetic algorithm," in *Proceedings of the International Conference on Information Systems, Technology and Management*, pp. 350-351, Berlin, Germany, 2009.

[21] P. R. Srivastava, K. Baby, and G. Raghurama, "An approach of optimal path generation using ant colony optimization," in *Proceedings of the TENCON 2009-2009 IEEE Region 10 Conference*, pp. 1–6, Singapore, January 2009.

[22] T. Yano, E. Martins, and F. L. de Sousa, "Generating feasible test paths from an executable model using a multiobjective approach," in *Proceedings of the Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pp. 236–239, Paris, France, April 2010.

[23] A. D. A. Neto and E. Martins, "An adaptive multi-objective heuristic search for model-based testing," in *Proceedings of the 2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pp. 193–200, Coimbra, Portugal, September 2018.

[24] T. Michael, F. Björn, and S. Lars, "The dresden ocl parser version 3.1," 2016, https://www.dresden-ocl.org/.

[25] Eclipse for java developers (2009), https://www.eclipse.org/--2008.

[26] K. Louzaoui and K. Benlhachmi, "An optimal approach of conformity assessment and robustness testing for object oriented constraints," *Journal of Theoretical and Applied Information Technology*, vol. 99, no. 23, 2021.

[27] K. Me_ert, N. Rotstan, C. Knowles, and U. Sangiorgi, "Jgap-java genetic algorithms and genetic programming package," 2012, https://jgap.sf.net.

[28] G. R. Lichtenstein, B. E. Sands, and M. Pazianas, "Prevention and treatment of osteoporosis in inflammatory bowel disease," *Inflammatory Bowel Diseases*, vol. 12, no. 8, pp. 797–813, 2006.

[29] C. López-Pujalte, V. P. Guerrero-Bote, and F. de Moya-Anegón, "Order-based fitness functions for genetic algorithms applied to relevance feedback," *Journal of the American*

Society for Information Science and Technology, vol. 54, no. 2, pp. 152–160, 2003.

[30] J. L. Dale, K. B. Beckman, J. L. E. Willett et al., "Comprehensive functional analysis of the Enterococcus faecalis core genome using an ordered, sequence-defined collection of insertional mutations in strain OG1RF," *mSystems*, vol. 3, no. 5, 18 pages, Article ID e00062, 2018.

[31] C. R. Reeves, "A genetic algorithm for flowshop sequencing," *Computers & Operations Research*, vol. 22, no. 1, pp. 5–13, 1995.

[32] https://irigs.iiu.edu.pk:64447/gsdl/collect/00electron/tmp/T09568ESEMS.html.