Class Testing From OCL Class Contract Specifications Using Evolutionary Multi-Objective Genetic Algorithms

Supervised By
Mr. Atif Aftab Ahmed Jilani
Assistant Professor, FAST NU

Co-Supervised By
Mr. Syed Muhammad Saqulain
Assistant Professor, IIUI

Submitted By
Rehan Frooq
(119-FAS/MSSE/F06)

Department of
Computer Science and Software Engineering
International Islamic University, Islamabad

International Islamic Univ.
CENTRAL
LIBRARY
ISLAMABAD.

Accession No TH-9568

MA/MSC
005
JIC

1 - Software portability

2. Software compatibility

DATA ENTERED

[illegible] 25/9/13

A thesis submitted to

The Department of Computer Science and Software Engineering,

International Islamic University, Islamabad

as a partial fulfillment of requirement for the award of the degree of

MS in Software Engineering

<u>**DECLARATION**</u>

I hereby declare and affirm that this thesis neither as a whole nor as part thereof has been copied out from any source. It is further declared that I have completed this thesis on the basis of my personal efforts, made under the sincere guidance of my supervisors. If any part of this report is proven to be copied out or found to be a reproduction of someone else, I shall stand by the consequences. No portion of the work presented in this report has been submitted in support of an application for other degree or qualification of this or any other University or Institute of learning.

Rehan Farooq (119-FAS/MSSE/F06)

Acknowledgements

This is an opportunity to thank **Allah Almighty**, our creator the lord of the heavens and the earth, who gave me courage to stand all the difficulties, and blessed me with all His blessings throughout my life. It is beyond any doubt, due to **His** blessings, that the targets, once seemed unachievable were attained successfully.

It was only due to, continues support and guidance of my research supervisor **Mr. Atif Aftab Ahmad Jilani (Assistant Professor FAST NU)** and co-supervisor Mr. **Syed Muhammad Saqlain (Assistant Professor IIU).** They were always there when needed and supported me in any possible way they could. I am really grateful to both of them and their contributions in my research are priceless, may **Allah** bless them both.

Acknowledge is also due to my family, wife and kids. I am what I am, at the moment due to my parents and siblings, and their tangible and moral support. Without the support of my wife I would not have been able to get enough time to work on my research, she and my kids sacrificed their time for me so that I may concentrate on my research. Big acknowledgement is due to my office **Bahria Town Pvt (Ltd)** and especially **IT Department**, they relaxed me whenever I required, offered me their resources and allowed me to complete my research.

Rehan Farooq (119-FAS/MSSE/F06)

Abstract

Trend of Software has been towards building bigger, more complex and highly reliable systems. These trends turn Software failures into fatal and causing catastrophic damages to human life and wealth. It obviously, becomes extremely important that we must thoroughly test software systems, to be safe prior to being actually used. Testing of UML Class models from their semiformal OCL specifications can help identification of defects early in the software life cycle. Current approaches suffer from inherent problems of exhaustive exploration of finite state machines (infeasible paths, exponential number of test sequences and uncertainty of completions of testing). Evolutionary algorithms can greatly help by optimizing the test sequences to get optimal coverage, minimal cost and higher quality.

Our new proposed approach can help improve the testing of UML model based software; by testing the conformance to semi-formal class operation contract specifications (specified in the form of OMG standard, OCL semiformal language). We achieve two main goals (1) Automation of testing process and conformance to standards, of current technique of test sequence generation, bridging the gap between the research and industry (2) Improvements in the state of the art approach through the application of Multi-Objective Genetic Algorithms (MOGA). Our Java based Testing tool, using our new approach, gives Test Engineers, choice of selecting better quality test sequences, optimized in terms of quality and coverage. Automation process makes possible the adaptation to changed class contract specifications in a dynamic environment.

TABLE OF CONTENTS

List of Figures and Tables

CHAPTER 1

INTRODUCTION

In this chapter, we present the background knowledge of our research including, unit-testing from OCL class contract specifications, automation of test sequence generation process and optimization of generated state-based test sequences using evolutionary approaches. In the same way, motivations and research objectives will be discussed in a precise manner. We will also formulate the problems faced in the area, in order to apply the current approach to the testing process. At the end of this chapter, the outline and flow of the thesis is explained.

1.1. Overview

This research targets testing of UML class Models from their Class contract specifications. It lies at the intersection of Model Based Testing, Specification Based Testing and Evolutionary Testing, which are subject-areas of Software Testing, area of Software Engineering. It specifically targets Optimization of Test sequences generated from state-based approach, for unit testing of class, from semiformal OCL class contract specifications. This introductory chapter gives the background of our work and its significance in the domain of software engineering and software testing. Software Testing is a discipline of Software Engineering which deals with the testing of the software to reveal errors and indicate the quality of the software. Modern trends in software engineering directly affect the software testing process. Test engineers and test teams today face the challenges of testing large scale systems that might require exponential time and resources while being built and tested. Changes in requirements are quite often, and has risk of wrongly elicited (the tacit knowledge) or ill documented requirements. All these factors point towards a strong need of automaton and optimization for testing approaches. Due to large scale of built software and dynamic stake holder requirements; manual testing of software becomes impossible [17]. Testing of software in turn becomes strong candidate of automation along with a need to figure out the ways by which we can efficiently test the software keeping within the limited budgets of time and cost. Many authors have worked towards automation and optimization of testing process as discussed in the literature review section, but there are still many grayi areas where there are questions that need to be addressed by the research.

Specification-based testing refers to the area of software testing where software is tested against its specification. It is a type of functional black-box testing where software is tested on its interfaces for the validation against the documented requirement specifications. This discipline 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. One of the biggest plus of this type of testing is that it allows building of testing environment for the software even before the existence of the software [1], [11], [14] and [18].

Model-based testing is a sub-area of Model-based development and Model drive engineering, where we represent software in terms of models. One of the famous modeling techniques used is Unified Model Language (UML) where software is modeled in the form of static structures (e.g. class diagrams) and dynamic structures (e.g. sequence diagrams). Model-based development lets the engineers to focus on the actual domain specific issues compared to technical issues of software development process itself. A big advantage of Model-based development and Model-based testing is the availability of tools support. Tools are available that can help engineers model software, transform software models from one representation to another and generate abstract test cases extracted from the software model. These abstract test cases can then be transformed into actual executable tests [11], [16].

Genetic algorithms are random search-based heuristics. They mimic the natural process of evolution; they are also referred to as simulated genetic algorithms. As the theory of evolution states, that living things get improved generation after generation and adopt better quality combinations of genes. While using genetic algorithm for a problem optimization the very first step is representation of the potential solutions in terms of chromosomes. Each chromosome consists of number of genes; genes are part of a potential solution to problem at hand. Together these genes and chromosomes form the population of possible solutions. MOGA tools execute Genetic algorithms, applying genetic operators on the input population.

The evolution process involves following steps:

- *Initialization* of the population, random or from some input.
- *Selection* of fittest individuals based on their calculated fitness values.
- *Reproduction* of the selected individuals.
- *Termination* of the evolutionary process based on selected criteria e.g. n number of generations or *f* target fitness values.

Reproduction involves application of the genetic operators based on probability.

- *Crossover* is a genetic operator where two or more than two solutions are combined to form resulting child solution. A number of techniques for crossover are available in the literature. The simplest is called "*single point crossover*", where a part of first chromosome and the remaining part of the second, disjoined at a point of crossover, is taken and combined to produce resulting child chromosomes.

- *Mutation* is the random change in part of a chromosome that results in a new individual with properties different from the parent. Depending upon the selected probability one or more of the genes can be change at random by the GA execution mechanism.

Traditionally GAs has been used as a search heuristic for finding optimal set of solutions to problems involving single objective. Recent advances in the field suggested usage of GAs for multi objective optimization [16]. In principle Multi Objective GAs are the same GA based tools, but the potential solutions are evaluated for multiple parameters and their fitness values are evaluated by multiple fitness functions. MOGA evolution process then involves comparison of the multiple fitness values of candidate chromosomes. Multi Objective optimization is particularly used with problems where no objective can be optimized without sacrificing the quality of the competitive objective(s). The solutions so generated are referred to as Pareto-Optimal solutions. Test sequence optimization involves trade-off between testing cost and achieved test coverage; hence the process is a strong candidate of Multi-Objective Optimization [17].

1.2. Motivation

Since the earliest development of computer program, software has come a long way and through many paradigms. It was journey form a few lines of computer instructions punched on machine readable cards to millions of lines of code to develop high end graphical user interfaces. The trends in the last three decades in software engineering have been to build bigger solutions to bigger and more complex problems, from a single user programs to multi-user, geographically distributed applications with multiple tiers and from an application affecting a few users to the application affecting the humans all over the world. Historical paper titled "No Silver Bullet" written by F.P. Brooks still holds even after 3 decades of its publication [20]. Brooks discusses inherent properties of software like Complexity, Conformity, Changeability and Invisibility.

As the software has grown, became more and more complex and ultra dependable; the need of finding and fixing problems, before the actual deployment and early in the Software Development Life Cycle has grown enormously. Over the past decades the trends in the software development have shifted from being considered as Art of individual programmer towards establishment of Engineering grounds and principles.

A very first consequence of the application of Engineering Principles to the world of Software Development was the thinking of software as a product as any other industry product. This raised questions about quality of the software and introduction of concept of quality.

Quality of software must be tested against the intended behavior as specified by the software requirement specifications. It raises major concerns Firstly, software requirements should be specified so that they could formally be tested against the actually developed software [1]. Secondly, requirement specification techniques should be understandable by software developers and should be close to programming syntax in order to be used in the industry (OMG's like Object Constraints Language OCL [21]). Thirdly, some techniques should be devised to map these requirements to the actual functionally of the software. Finally, problem domain of testing has unlimited testing combinations for different input variable values so testing sequence should be figured out to the reveal most of the possible errors in the software implementation.

1.3. Problem Statement

Model Based Testing involves automation of testing process. Building a model of the System Under Test (SUT) and then; generation, execution and evaluation of Test cases for SUT. Operation contracts specify the class behavior in terms of invariants, pre and post conditions, these class contracts are bindings that SUT must conform to. An obvious advantage of using class contracts is that they can be written in form of semiformal OCL constructs which are more precise compared to the natural language specifications and also can be easily converted to a machine readable form. The survey of literature reveals that class contracts have potential of revealing the test sequences for the unit testing of classes [1], but to-date very little work has be done in this direction. State of the art approaches also lack automation and conformance to industry standards.

Search based optimization algorithms on the other hand have been employed widely in the field of MBT but to date there is no evidence of their application for test sequence identification from class contract specifications. Optimization techniques are promising for optimization of number and quality of test sequences by overcoming the state space explosion problem.

1.4. Research Objectives

Our research targets improvement of the current OCL class contract specification based test sequence generation process [1] in multiple ways by:

a. Applying current state of the art test sequence generation technique to the industry standard OCL class contract specifications.

b. Automation of the current technique of test sequence generation from OCL class contract specifications.

c. Improvement of the current specs based test sequence generation approach by application of search-based techniques of Evolutionary, Multi Objective Genetic Algorithms

    a. Optimizing the test Coverage achieved by the generated test sequences.

    b. Minimize the number of infeasible test sequences.

This research answers the following questions:

- How we can improve the Unit Testing of Class Models using OCL class Contract specifications in terms of compliance to industry standards and automation of the Test Sequence Generation Process?
- How state of the art techniques of Optimization (Evolutionary Genetic Algorithms) can be applied to the problem of determination of Test Sequences based on OCL Class Specifications to achieve reduction in number of infeasible test sequences and improvement in test coverage?

1.5. Thesis outline

The rest of the thesis is organized as follows

- **Chapter 1** describes the overview of Software Testing and Test Sequence Generation, approaches of optimization using GA and MOGA, problems with current approaches, motivation and the objectives of the thesis.
- **Chapter 2** presents the literature review related to test sequence generation in association with OCL, UML Models, GA and MOGA. In this chapter we discuss numerous Test Sequence Generation techniques. Moreover, we present identified limitation in the literature. At the end of the chapter, we present analysis of literature in the form of a table.
- **Chapter 3** defines the proposed approach based on the found limitations in the literature. In this chapter, we present flow of proposed approach. We discuss the different modules of the proposed approach. We also discuss the Genetic Algorithm (GA) and the different operators of GA. Furthermore, we present algorithm for the proposed approach that how we program the proposed model.
- **Chapter 4** presents a detailed overview of the results obtained after implementation of the proposed approach. In this chapter, we present the implementation of our approach in Eclipse IDE for Java as development tool, Java 1.6 (Java 6) as programming language. Use of industry standard DresdenOCL parser [22] for parsing OCL operation contracts, Java Genetic

Algorithm Package (JGAP) [23] for MOGA execution, JUnit [26] for unit testing of Class Under Test (CUT) and Java Reflection API [27] for analyzing run-time behavior of the CUT. Finally, the claim is validated by comparing the proposed approach with current approach in the literature.

- **Chapter 5** provides conclusion of the current research work. This chapter also presents the future work direction to carry out further work in such an important research area.

CHAPTER 2

LITERATURE REVIEW

It is notable that the work is diverse in nature and spans across boundaries of the areas of software engineering. We have divided the review in sections as the work on test sequence generation (from state and UML Models), test sequences generation from formal specification (especially form operation contracts) and test sequence optimization (by single and multi objective approaches).

2.1. Test Sequences Generation

Generation of test sequences (synonymous to test cases) is one of the toughest tasks for a test engineer. This testing phase involves trade-offs between number of test cases and the desired test coverage, number of test cases and available resources, quality of test cases and achieved coverage etc. This test sequence generation process can be quite tiresome if done manually, so literature has quite a lot of work concerning automation and optimization of the process.

2.1.1. Test sequences from State Models

Ruilian Zhao et al [3] aim to develop the infrastructure of automatic test data generation for EFSM models that produce real data to trigger feasible transition paths. It also provides empirical results on efficiency analysis of test data generation for a set of state-based models. In this paper, a GA-based system is presented to automatically generate test data for feasible transition paths in EFSM models.

Karnig Derderian et al [13] present an approach for automated Unique Input Output (UIO) sequences generation for finite state models. They take sequence generation problem as a search problem and generate test sequences based on Genetic Algorithms

(GA). They use 11 real and 23 randomly generated FSMs as proof of concept experiment. They also state that the problem of test sequence generation from an FSM is an NP-Complete problem. The presented experimental results show that GAs give result between the ranges of 62% better to at least as good as random search. They also propose a new fitness function for evaluating fitness values of UIO test sequences and claim that it is performance wise better than the previous approach. They suggest that at small FSMs random search seems to outperform GA but for bigger FSM models GA are a far better approximation.

2.1.2. Test sequences from UML Models

UML diagrams model static and dynamic aspects of a system, techniques found in the literature in general use one of the static diagrams to represent the static structure and one of the dynamic diagrams to represent dynamic behavior of the software, in order to generate test sequences / test cases.

S. Asthana et al [6] have given an approach for generating test cases from class and sequence diagrams the claim is that this is the novel approach which uses test cases from class and sequence diagrams without transforming them into any intermediate model. The approach claims that use of any intermediate form is avoided by the approach from specification model to actual SUT, but XMI itself seems to be an intermediate form used for representation of the model.

Chen Mingsong et al [12] present an approach of test case generation from UML activity diagrams. In their approach they compare the dynamic behavior of the activity diagram to the actual program execution and in this way the activity diagram behaves as a directed graph. They use three test selection criteria activity coverage, all transition coverage and simple path coverage. Code instrumentation is used for recording test data and the test logging statements are inserted into the program itself. This approach is a white box testing approach because it needs access to the program source for testing.

2.1.3. Test Sequences from Software Specifications

Atul Gupta [1] discusses an approach where class contracts are used to test class method interactions. The approach is state based approach. Using an abstract state configuration of class and initial abstract states, reachable states are incrementally generated by searching for the methods which can be invoked in the current state and resulting abstract states are computed. It lacks automation and syntax used does not conform to the industry OCL standards and fails even to get parsed by standard OCL parsers. The approach uses AFS traversal to generate test sequence paths, hence faces inherent problems of finite state traversal which we discuss in detail in our pitfalls section.

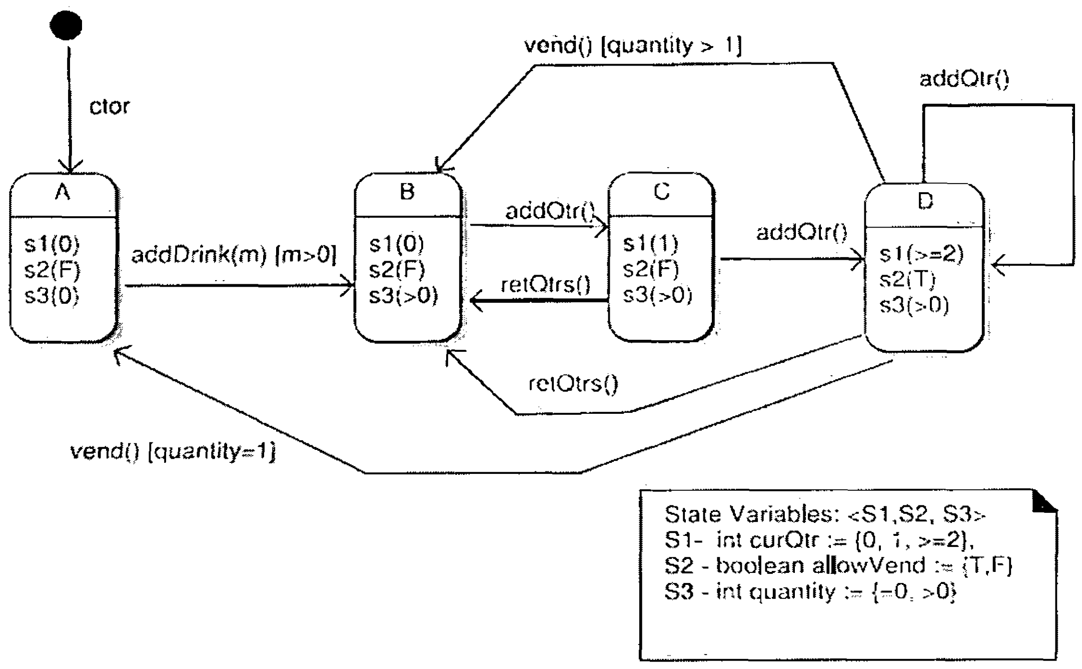


Fig.1.1. Generated abstract state model for the class CoinBox [1].

This is the core reference paper used by us and hence we give a brief over view of the approach here. Author has used the traditional searching approach for path traversal of finite state machines and all transition coverage is used as sequence path generation. A specification based testing approach is proposed, which uses class contracts specified in the form of OCL constraints (class invariants, pre conqditions and post conditions). They build an abstract state configuration for the class under test, for each initial abstract state, corresponding reachable states are incrementally generated by traversing and searching for the methods which are invoke-able in the current state and resulting abstract states are generated. Author argues that state of an object, being specified by values of its variables can lead to state explosion and hence notation of "abstract states" is introduced having abstract object variable values.

Applying Transition Tree Coverage

Abstract State Machine → Test Sequences

Testing sequences were generated using Transition Tree coverage and Modified Transition Tree coverage (by including additional test by for invalid inputs). The thing which is lacking in the approach is that it is still not automated (author himself mentions that in the conclusion section) and no tool has been suggested for automating the suggested process.

T Miller and Paul Strooper [11] present a case study on specification based implementation testing frame work. They have used Possum animation tool and Sum specification language for modeling and specification of GSM 11.11 standard of mobile communication. They claim that the framework gives almost equal performance compared to BZ-Testing tools and more cost effective than manual testing. Authors suggest stepwise generation of a directed graph and then paths through that graph are the test sequences.

Marie-Claude Gaudel [15] presents an approach for generation/ selection of test data from the formal specifications. An exhaustive test set based on the formal specifications and their correct implementation is proposed. After that selection of a finite test set is proposed based on domain specific selection hypothesis. Author presents result of case studies of application of the approach to algebraic specifications in the form of LOTOS based specifications of ISO OSI based protocol specifications. A big constraint in the application of this approach may be of manual work involved in order to decide to

"Selection Hypothesis" that varies from domain to domain and specifications to specification.

Planning and execution of tests involves the analysis of the functionality of software (functional specs), what are the inputs and outputs of the software and its execution environment. This process is difficult, time taking and technically sophisticated. Role of a tester requires him/her to have programming skills, grip on formal languages like OCL, mathematical theory of graphs and good understanding and comprehension of computer algorithms [10].

Literary survey reveals that most of state of the art research targets test sequence generation using UML static diagrams(class diagram) and UML dynamic diagrams (Sequence diagram and State Charts). UML diagrams are not sufficient enough for specifying complete class behavior, most accurate details of a class are revealed from the OCL class specifications in the form of OCL Class Contracts [1].

Test sequences generated using the OCL class contract specifications using state-based technique suffer from their inherent problems including infeasible-paths and exponential number of generated test sequences. In this research we try to figure out solution to these problems besides have automated and optimize the test sequences generated from the OCL Class contracts specifications. Multi Objective Genetic Algorithms are used to overcome the issues by their power of search based multi objective optimization as discussed in [2], [4], [7] and [9].

2.2. Test Sequence Optimization

Shukatl Ali et al [7] preset a systematic review of search-based test case generation techniques. The plus is a comparison of different Meta Heuristic Search (MHS) algorithms being employed in search-based testing of software. They have assessed 450 papers out of 6 research repositories. They conclude that Genetic Algorithms are promising for problem solving in the domain of software testing.

2.2.1. Single Objective Optimization

Mark Harman et al [4] propose three search-based algorithms for test data generation and preset the result of a case study for the application of their approach. The claim made by authors is that their approach can maximizes the coverage and minimizes the number of test cases generated. The size of the software considered for case studies is as big as 144 lines of code, which might be good for a proof of concept.

Andrea Arcuri et al [5] focus on comparison of 3 test automation strategies namely Random Testing, Adaptive Random Testing and Search-based testing using Genetic Algorithms and present their results. They present a comparative analysis of the approaches and present the results of experiment on 3 SUTs.

S.K. Prasad et al [8] present GA based approach for test data generation and they present their algorithm that takes the user input variables and using GA generates test data. They claim that GA outperforms random testing on time measures [7]. In another paper S.K. Prasad et al present another search-based test sequence generation technique using Ant Colony optimization algorithm where "Ants" are used to explore CFG to find optimized test sequences.

Compared to competitive optimization techniques, GAs, instead of searching a solution by heuristic search methods, start with a random set of possible solutions and then improve the solutions by simulation of evolutionary processes of crossover, mutation and selection. This process is repeated generation after generation. That way an optimized set of solutions is guaranteed, which can always be improved further by subsequent GA implementation, as the optimization techniques give optimal solution(s) because exact solution is not available [3].

GA techniques are independent from the problem domain; this is quite helpful for general purpose optimization of the problem, because the GA implementation takes encoded representation of the problem and yields the optimized results irrespective of the problem at hand. Being random search algorithms, they avoid convergence to local minima and the solutions are quite evenly distributed across the problem domain.

2.1.2. Multi-Objective Optimization

Thaise Yano et al [2] present an approach of test sequence generation using Evolutionary Algorithms. The claim that search based approaches till then had been mostly proposed for white-box testing. The paper presents, an evolutionary approach for test sequence generation from a behavioral model, in particular, EFSM. A multi-objective evolutionary algorithm, M-GEOvsl adopted from M-GEO is used, that can consider two objectives: to search for a test sequence that covers a target transition, as well as to minimize the length of this test sequence [2]. Authors present an approach of test sequence generation using Evolutionary Algorithms. They claim that search based approaches till then had been mostly proposed for white-box testing. The paper presents, an evolutionary approach for test sequence generation from a behavioral model, in particular, EFSM. Problem of Infeasible paths generations is covered by executable model. Transition of interest coverage criterion is applied using Evolutionary Algorithm. System is modeled in form of EFSM. Challenges listed by the authors while generating test for EFSM. An Evolutionary Algorithm is also proposed, based on Pareto optimality. Each solution is non-dominating, that is, it can't be improved in any objective without causing degradation in at least one other objective. Future work of the authors suggest improvements like addressing the limitation of the approach when there are no slices of a model are found and validation of the approach is demonstrated by an experiment but they sate that they are carrying out further experiments for the validation of the approach [2].

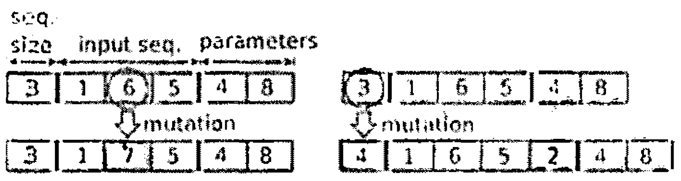


Fig. 1.2. Mutations in M-GEO$_{eval}$ [2].

Multi Objective Genetic Algorithm (MOGA) go one step further, they support optimization for multiple objectives, in our case optimization for two objectives, minimize the number of test sequences and maximize achieved test coverage of the test sequences is required[2].

MOGAs have a very good support by open source tools like Java Genetic Algorithm Package (JGAP), JMetal (a multi-objective GA implementation tool) and Java API for

Genetic Algorithms (JAGA) [12],[13] and [14]. These and similar tools, being used in the industry and research, it makes them more practical to bc used for the practical test sequence optimization for industry usage.

2.3. Literature Evaluation

Approaches and techniques in the literate has different problems here we discuss these identified problems as found in the literature.

2.3.1. Conformance to Standards

Class contract based test sequence generation technique found in the literature [1] does not conform to industry standard OCL syntax so it makes the process impractical, while being adopted by industry practitioners. Due to the same reason current technique lacks automation. The first phase of the research focuses on adopting the technique to work on standard OCL syntax. We take standard OCL syntax specs and apply the test sequence generation technique to get the output test sequences. This kind of test sequence generation approach is state-based as discussed [1].

2.3.2. Lack of Automation or difficulties while automating

Approaches found in the literature either don't provide any automation at all (assuming the input in a predefined state) or Fail to comply with the state of the art industry standards like e.g. deviation for the standard syntax It makes it hard for test engineers to used these techniques Software requirements and hence specification are quite often volatile, automation can be a great help to regenerate the test sequences from new specifications [1], [2], [4], and [15].

2.3.3. State-based problems

Almost all the approach used in the literature use Graph/State Machines as an intermediate form of representation for the software before generation of test case / sequence [1], [2], [6], [9], [12], [13] and [14]. These State-based approaches suffer from inherent state space exploration problems. A large number of possible test sequences may require exponential time and effort for the testing process itself. Unfortunately resources and time are limited for the Software Development Lifecycle (SDLC). Many of the state-based generated test sequences might be Infeasible, repetitive, reoccurring possibly several times or might not be required at all. It is not practical and, in general, impossible to asses all the possible test sequences of program flows due to effort and time required for execution. There is always a tradeoff between number of generated test sequences (cost) and the achieved test coverage (coverage). It is quite difficult for a machine to evaluate all test sequences within a reasonable amount of time. Exhaustive testing of all the test sequences is impossible.

2.3.4. The Need and Potential for Optimization

Being state-based the technique suffers from inherent problems of state-based test sequence generation techniques [2], [3] and [4], and can be improved by applying search-based optimization techniques. Multi Objective GA's are promising for the improvement where we can remove infeasible test sequences using multiple fitness functions to achieve maximum test coverage in minimum number of test sequences.

The next phase is to optimize the generated test sequences using Evolutionary Genetic Algorithms using a multi objective approach where we have two conflicting objectives first to minimize number of test sequences and second to maximize test coverage of generated test sequences.

Approach discussed by [2] for test data generation using GA but a similar approach can be used in our case for generating test sequences using Multi Objective Genetic Algorithms.

2.4. Summary

Table 2.1 summarizes the comparison of different approaches found in the literature along with the parameters most related to our research.

Table 2.1 Summary of Literature Review

Authors	Automation	Specification Based	Coverage	State Based	Optimization	Multi Objective
Atul Gupta [1], Springer (2010)	Automation is hard due to non-standard OCL syntax	Yes, from OCL specifications	-Transition Tree Coverage	Yes, suffer from State-based problems	X	X
Marie-Claude Gaudel [14], Springer (2001)	Semi Automation	Yes, LOTOS based proof of concept on Possum.	-All paths coverage	Yes, test sequences are generated from directed graph	X	X
Thaise Yano et al [2], ICSTW, IEEE (2010)	Partial automation does not discuss in which form the model will be taken.	X	-Target Transition Coverage	EFSM is used as an intermediate form	Yes, an Optimization Algorithm, which is strictly not GA based.	Yes
Mark Harman et al [4], ISCTW, IEEE (2010)	X	X	X	Yes an EFSM based representation is used	Three Test data optimization algorithms are proposed	X
S. Asthana et al [6], Springer (2010)	Automation without using an intermediate model.	X	X	Yes , and claim to have avoided state space explosion because their model is executable	X	X
S.K. Prasad et al [8], ICISTM, Springer	Claim automatic approach for generating test	X	X	X	Yes optimization through single	X

(2009)	data				objective GA	
S.K. Prasad et al [9]. ICISTM. Springer (2009)	Automation of test sequence generation process is claimed.	X	-All state coverage	Yes. FSM is generated.	Yes .Ant Colony based Optimization	X
M. Prasannan and K.R. Chandran [10]. ICSRS (2009)	Automation of test case generation is claimed.	X	X	General Tree and Tree Structure are built and depth-first search gives test sequences	Crossover step of GA is used. but optimization is not mentioned.	X
Chen Mingsong et al [12]. ACM (2006)	Yes from UML Activity Diagram	X	-Activity -Simple Path -All Transition	Yes. UML Activity Diagram as Directed Graph.	X	X
K. Derderian et al [13]. ACM (2006).	Yes	X	X	Yes. Approach is specifically for FSMs	Yes. GA based optimization	X

CHAPTER 3

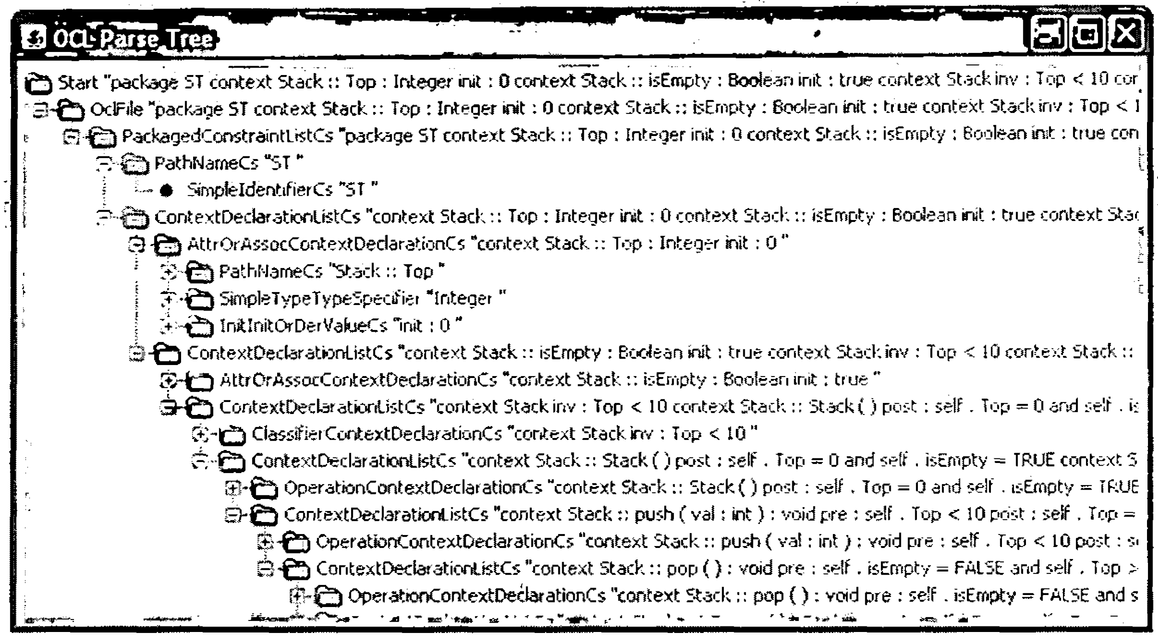
PROPOSED APPROACH

Our proposed approach caters for the limitation of the current approach by using the OMG's standard OCL syntax and automation of the test sequence generation. In order to improve testing effectiveness we apply a Multi-Objective approach using MOGA where Optimization for test coverage and validity of the test sequences is a concern. Our goal is to produce test sequences which are most effective in identification and revealing of software implementation problems.

Our approach improves the previous approach in a number of ways; here we explain the actual functionality of our approach and the advantages achieved. Our new approach is divided in two main phases in first phase standard OCL parsing is done on the input OCL class contracts and an Abstract Finite State Machine is generated using the rules specified by the previous approach. Second phase involves optimization of state-based test sequences, generated from the source AFSM using multi-objective GA.

3.1 Parsing of Class Contracts and Generation of Abstract Finite State Machine

We use standard OCL syntax and build the test sequences from the generated Abstract Finite State Machine. We use standard OCL parser [22] for generating OCL pares tree of input class contracts. This parser is frequently used with Eclipse IDE for Java [25] for parsing of OCL constraints on UML Models. This parser is responsible for generating OCL parse tree from the textual OCL class contracts.


OCL Parse Tree

```
Start "package ST context Stack :: Top : Integer init : 0 context Stack :: isEmpty : Boolean init : true context Stack inv : Top < 10 cor
  OclFile "package ST context Stack :: Top : Integer init : 0 context Stack :: isEmpty : Boolean init : true context Stack inv : Top < 1
    PackagedConstraintListCs "package ST context Stack :: Top : Integer init : 0 context Stack :: isEmpty : Boolean init : true con
      PathNameCs "ST "
        ● SimpleIdentifierCs "ST "
      ContextDeclarationListCs "context Stack :: Top : Integer init : 0 context Stack :: isEmpty : Boolean init : true context Stac
        AttrOrAssocContextDeclarationCs "context Stack :: Top : Integer init : 0 "
          PathNameCs "Stack :: Top "
          SimpleTypeTypeSpecifier "Integer "
          InitInitOrDerValueCs "init : 0 "
        ContextDeclarationListCs "context Stack :: isEmpty : Boolean init : true context Stack inv : Top < 10 context Stack ::
          AttrOrAssocContextDeclarationCs "context Stack :: isEmpty : Boolean init : true "
          ContextDeclarationListCs "context Stack inv : Top < 10 context Stack :: Stack ( ) post : self . Top = 0 and self . is
            ClassifierContextDeclarationCs "context Stack inv : Top < 10 "
            ContextDeclarationListCs "context Stack :: Stack ( ) post : self . Top = 0 and self . isEmpty = TRUE context S
              OperationContextDeclarationCs "context Stack :: Stack ( ) post : self . Top = 0 and self . isEmpty = TRUE
              ContextDeclarationListCs "context Stack :: push ( val : int ) : void pre : self . Top < 10 post : self . Top =
                OperationContextDeclarationCs "context Stack :: push ( val : int ) : void pre : self . Top < 10 post : s
                ContextDeclarationListCs "context Stack :: pop ( ) : void pre : self . isEmpty = FALSE and self . Top >
                  OperationContextDeclarationCs "context Stack :: pop ( ) : void pre : self . isEmpty = FALSE and s
```

Fig. 3.1. Sample Partial Parse-Tree of OCL Operation Contract for Stack Class

After generation of parse tree is the process of semantic analysis of the output parse tree and construction of domain specific objects in Java. Our OCL parse tree processor transverses the parse tree and extract the Objects corresponding to the domain concepts of OCL semantics.

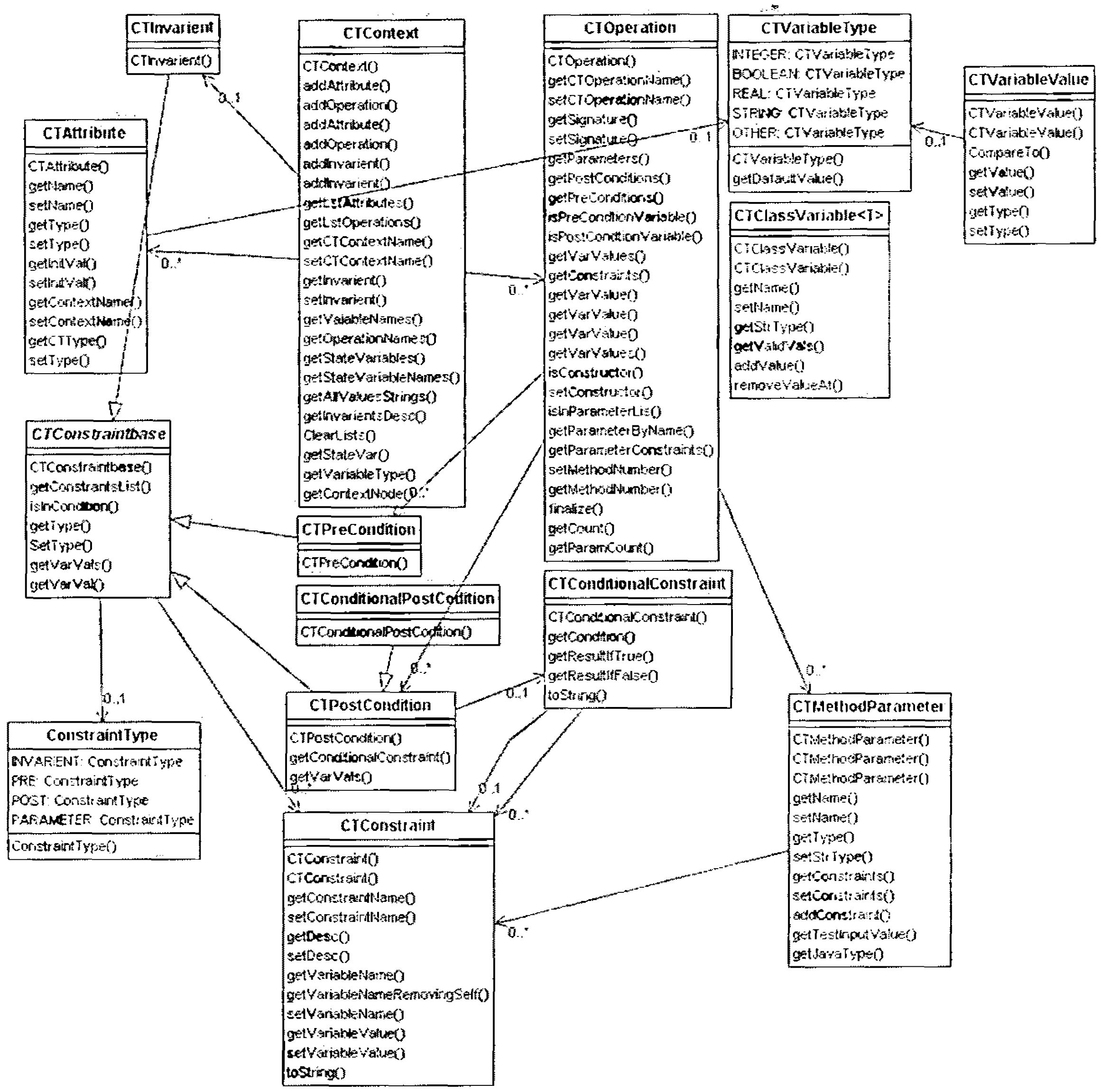


Fig. 3.2. The Class Diagram of Mapping Objects of OCL Operation Contracts

After generation of parse tree the next step is of constructing the abstract finite state machine applying the rules used by [1]. The abstract state model of the software from specification is created starting from the class constructs. For each constructor a new initial state in the Abstract Finite State Machine is created. We then dynamically create all the resulting states from the initial state onwards.

TH-9568

A deviation here is that previous approach suggest. using transition tree coverage criterion i.e. test sequences are identified along with the simple paths. Simple paths coverage misses the self reference transitions and it is quite possible that a method might fail on subsequent invocations as the subsequent calls might bring the object in as state (due to implementation faults) that it may behave anomalously; even the specifications may suggest some other behavior.

But in case we have self transitions to a state then it might skip a valid step in the sequence of method calls. So it is better if we have 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.

3.2 Coding of Test Sequences in Chromosomes and Optimization through MOGA

After buildup of the abstract finite sate machine the next phase is of generation and optimization of the testing sequences. This phase involves coding of the test sequences in tool specific Chromosomes, execution of MOGA and selection of best fit test sequences after evolution.

3.2.1 Coding of Solutions in Genes and Chromosomes

We have devised a coding scheme where a potential solution (Chromosome) comprises of Transition (Test Transition) from the built Abstract Finite State Machine. Each Test Transition represents a transition in the Abstract Finite State Machine with additional feature to be automatically executable on a class under test by calling the method represented by this transition.

Generally solutions or chromosomes are coded in the form of binary string values representing the potential solutions to the problem at hand. But our coding scheme is not a binary coding were each gene is coded in terms of a binary representation. Modern day tools allow usage of custom, user defined genetic coding. Especially while working within Object Oriented language like Java, where everything is in essence an Object, we get infinitely many options for vales of each Gene.

So a Chromosome of length n will have n Test Transition objects (genes). The JGAP tool used by us allows specifying a mechanism of returning custom random genes values while population is evolved for the purpose of mutation. To tell the MOGA implementation system how to get random values we attach a mechanism which returns random transitions from the generated finite state machine. Here firstly a random state is picked out of all state of the finite machine, after that one of the outgoing transitions is selected at random. A potential chromosome in our solution set can be visualized as:

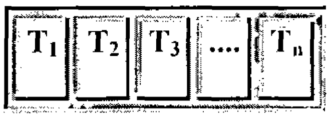


Fig. 3.3 A Chromosome of length n, in our coding scheme.

Each Transition Ti in the coding scheme, contains reference to an, initiating state (Transition from state) from which that transition originated and a reference to a terminating state, to which that transition is leading.

Where n is the length of the chromosome and T_i is the ith Transition in the test sequence and i = 1,2,3,...n. during the MOGA optimization mutation and crossover is applied on the genes. While during mutation the changed gene value is randomly selected transition from the built Abstract Finite State Machine.

3.2.2 The Multi Objectives

The test sequence generation process should be efficient enough to reveal the problems in the implementation. In order to get quality test sequences we use two objectives they are not totally in a conflict but optimization for one might decrease the fitness of the other objective. We have following two objectives while optimizing the test sequences, it should be noted that our aim here is to get the test sequences those are more revealing and uncovering the problems in the class implementation.

3.2.2.1 Optimize Coverage

While testing we are interested to reveal 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. We can only have as improved class test coverage as possible so that we are sure of a level of the quality of our testing process. So first of the two objectives we have 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.

3.2.2.2 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 the question arises of getting test sequences which are in sequence according their place in the finite sate model. Our second of the multi objectives 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 over all coverage achievable by that solution.

3.3 The Genetic Evolutionary Process

The Evolutionary process in our approach is completed in the following steps; this genetic evolution of chromosomes is done automatically by Java Genetic Algorithms Package (JGAP) [23], but as directed by our approach:

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. While this way we can save the efforts involved for traversing through the finite state machine but a comparatively better way is to get the initial pool by exhaustive search of the AFSM. Because, on taking the first kind of population, there is a possibility of evolution of the population towards local maxima.

3.3.2 Selection for Reproduction

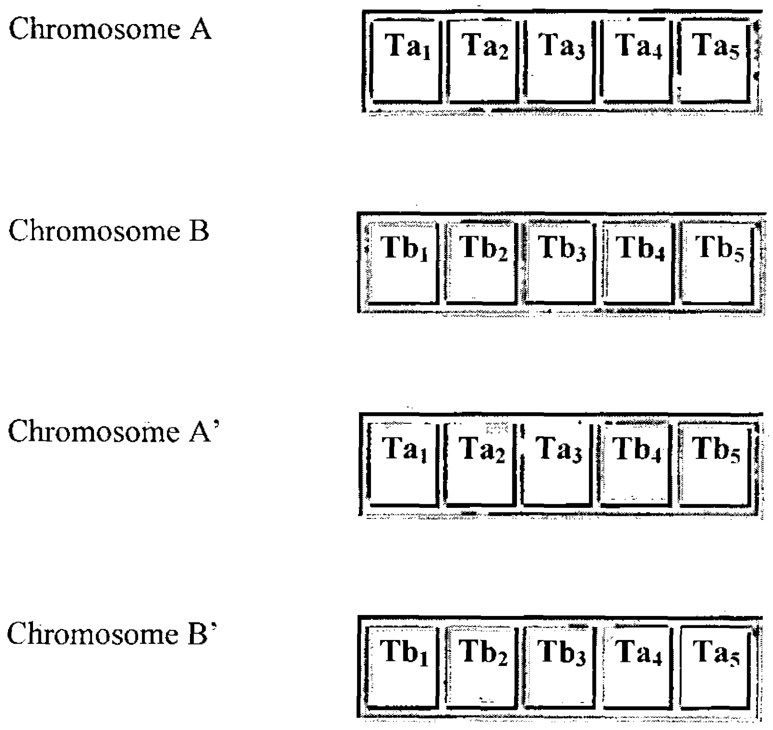
Process of selection involves selection of fittest individuals for mating in the next population. Here each gene is passed from the genetic Evolution tool to our fitness function evaluator and is then assigned fitness values based on our fitness functions.

3.3.3 Reproduction of Population

Population crated in step one under goes Genetic processes of Crossover and Mutation and gets evolved over generations. After each generation, chromosomes are assigned fitness values according the fitness functions.

3.3.3.1 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.

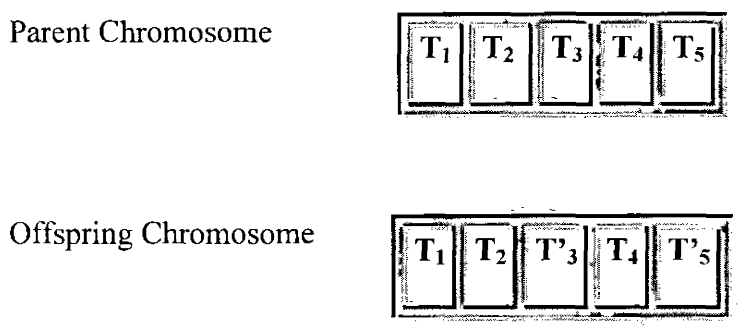


Chromosome A
Ta_1 Ta_2 Ta_3 Ta_4 Ta_5
Chromosome B
Tb_1 Tb_2 Tb_3 Tb_4 Tb_5
Chromosome A'
Ta_1 Ta_2 Ta_3 Tb_4 Tb_5
Chromosome B'
Tb_1 Tb_2 Tb_3 Ta_4 Ta_5

Fig. 3.4 Sample crossover process, Chromosomes A&B are changed to A'&B'.

3.3.3.2 Mutation

In this operation value(s) of Genes are mutated based on the mutation probability and resulting chromosomes are constructed. Number of genes changed during the process, depends on selected level of mutation and selected probability of mutation. Here some of the Test Transition objects in the target chromosome are replaced with randomly selected values form the AFSM. Our random transition selection mechanism plugs in with the evolution tool and provides it random transitions when required for mutation purpose.



Parent Chromosome
T_1 T_2 T_3 T_4 T_5
Offspring Chromosome
T_1 T_2 T'_3 T_4 T'_5

Fig. 3.5 Sample mutation process. Genes T_3 & T_5 of Parent Chromosome mutated to T'_3 & T'_5 to create Offspring Chromosome.

3.3.4 Termination Condition

Termination criteria in a genetic evolution process can be of two types, firstly when we evolve the population to reach a specific amount of fitness values and secondly where we evolve the population to a specific number of generations. Second termination can be used if we are sure of the required quality of the chromosomes, but while optimizing test sequences unfortunately, that is not the case. So we select the termination criterion of evolving the population, 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. The MOGA based tools use these user defined fitness to assess the quality of solutions (the chromosomes). The simulated genetic process of evolution, assigns the fitness values to the chromosomes for each generation and after application of genetic operator only fittest chromosomes are selected for subsequent generations. For test sequence optimization, we have devised the following fitness functions:

3.4.1 Calculate Fitness By Coverage

Calculates the coverage of current chromosome by the number of transitions covered and assigns the fitness value according to the following algorithm

$$Coverage\ Fitness\ (CF) = \sum_{i=1}^{n} (coverage\ weight\ for\ call\ sequence)i$$

Description of calculation of coverage weights is

- If a transition is covered once chromosome is given additional positive weight-age, it rewards a chromosome for covering a transition.
- If a transition is not covered at all by a chromosome it is given additional negative weight-age, it reward a chromosome negatively for not covering a transition.

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

The fitness value for a chromosome by coverage can be calculated by the following pseudo code:

```
Initialize CF:=0, wCoveredOnce, wCoveredTwice, wCoveredMoreThanTwice
For each Chromosome c in the current population
        For each Gene g in c
            If g occurs once
                CF = CF + wCoveredOnce
            End
            If g occurs twice
                CF = CF + wCoveredTwice
            End
            If g occurs more than twice
                CF = CF + wCoveredMoreThanTwice
            End
        End
    Set coverage fitness of c equals CF
End
```

wCoveredOnce, wCoveredTwice, wCoveredMoreThanTwice are, problem specific arbitrary weight-ages, for at least one state coverage, a state covered twice and a state covered more than two times.

3.4.2 Calculate Fitness By Test Sequence Order

This fitness function assesses the fitness of a solution chromosome by assessing how much that particular solution is in order according to the generated AFSM. A better test sequence will be more in order than its competitive test sequence. We get fitness value as weighted sum of all individual fitness values of each gene of a chromosome.

Mathematically the fitness by sequence validity for a chromosome is calculated as

$$Fitness\ Order(OF) = initial\ state\ weight + \sum_{i=1}^{n}(sequence\ weight\ for\ call\ sequence)i$$

Description of the weight calculation for test sequence order fitness is

- Initial state weight, if the first gene of the chromosome has an initial state of AFSM as from state then this weight is added else skipped.
- Sequence weight for call sequence, we calculate the quality of chromosome by the sequence of method calls and reward each chromosome by following formula
    - If any of the method calls (genes) is in a valid sequence then a positive weight is added to the second fitness value.
    - If any of the method calls (genes) is not in a valid sequence then a negative weight is added

The fitness value for a chromosome by validity can be calculated by the following pscudo code:

```
Initialize OF:=0, wSState, wInSeq, wNotInSeq
```

```
For each Chromosome c in the current population
    If c starts with an initial state
    OF = OF + wSState
    End
        For each Gene g in c
            If g is in sequence
            OF = OF + wInSeq
            Else
            OF = OF + wNotInSeq
        End
    Set order fitness of c equals OF
End
```

wSState, wInSeq and wNotInSeq are, problem specific arbitrary weight-ages, for starting with initial state, being in sequence and not being in sequence respectively.

3.5 Expected Benefits of Proposed Approach

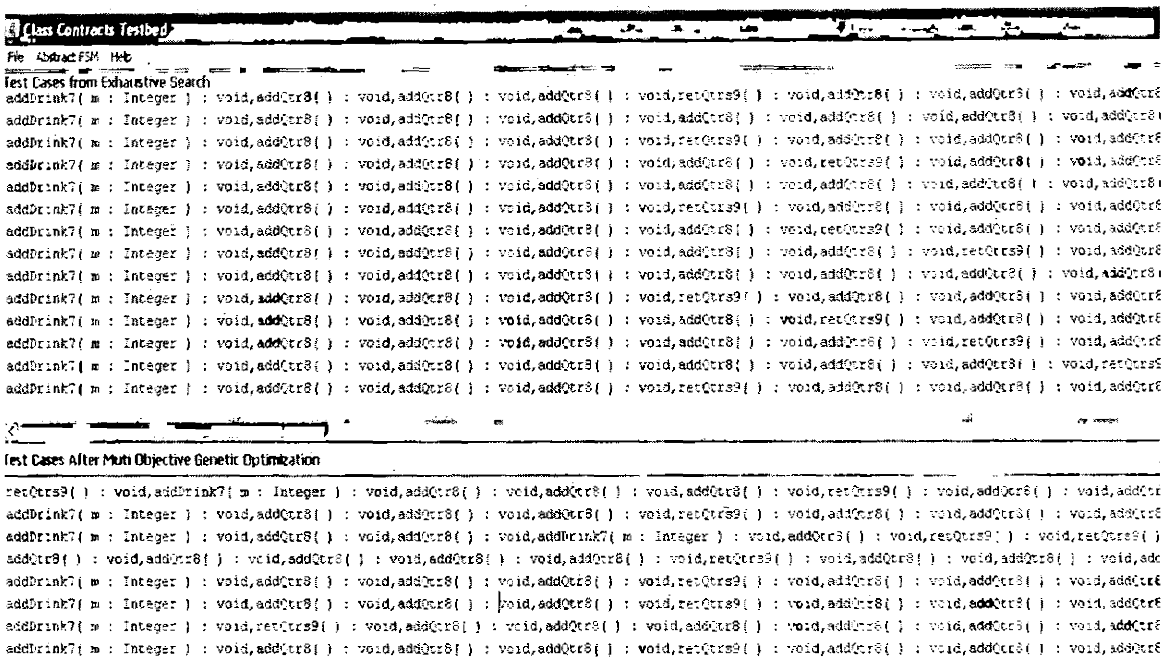
Proposed approach is expected to give following benefits over current approach of test sequence generation and optimization.

3.5.1 Adopting to Standards

The Object Management Group (OMG) has clearly defined specification for standard OCL syntax. As this thesis is written the current version is 2.3.1 as of January 2012, which is available on OMG's website for download. When OCL class contract, presented in the literature, is compared to the industry standard OCL syntax; it is revealed that, current approach deviates from the state of the art OCL used in the industry. As a matter of fact no standard OLC parser accepts the syntax used in the literature. Current syntax is more C++ like which is not acceptable as OCL syntax according to the OMG OCL specifications [22]. Since the approach deviated from the standards, it was quite unlikely to be adopted by the industry practitioners. In order to build our testing tool we observed that OCL syntax used by current approach fails to get parsed by standard OCL parsers that follow OMG's OCL class contract specification syntax. So we took the OCL Class contracts in standard OCL format and then applied current approach to it for building Abstract Finite State Machine (AFSM). Then we generated the test sequences by traversal of AFSM, beginning from the start state. We call these test sequences as raw test sequences because they suffer from the state-based path search problems. Now our approach is able to generate test sequences directly from standard OCL.

3.5.2 Automation

Conformance to standards provides the benefit of automation for the process of test sequence generation. Reading OCL class contract specifications, we automatically construct OCL parse tree. After that our tool does semantic analysis of constructed OCL parse tree and applying the rules defined in the literature build corresponding AFSM automatically. Next step is automatic generation of the raw test sequences from exhaustive search of AFSM, these test sequences can be directly used by the test engineers if they think raw test sequences test sequences are good enough and can be used without optimization. In case when test engineers decide to go for Multi-Objective GA based optimization for the test sequences, our tool automatically run MOGA over the raw test sequences selecting a random population out of them. This way the process of generating test sequences from standard OCL is automated all the way to the MOGA optimized test sequences.



Test Cases from Exhaustive Search

```
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
```

Test Cases After Muti Objective Genetic Optimization

```
retQtrs9( ) : void,addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addDrink7( m : Integer ) : void,addQtr8( ) : void,retQtrs9( ) : void,retQtrs9( )
addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,add
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
addDrink7( m : Integer ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr8( ) : void,retQtrs9( ) : void,addQtr8( ) : void,addQtr8( ) : void,addQtr
```

Fig. 3.6. Partial view of automatically generated, exhaustive search-based and MOGA-based optimized test sequences.

3.5.3 Optimization of Test Sequences

The new proposed approach is a novel approach that uses multi-objective GA for test sequence optimization, using an initial population of randomly selected exhaustive search test sequences. State of the art approaches found in the literature use random stochastic initial populations. By nature of MOGAs, use of complete random sequences, gives a big chance of getting the population evolved in a negative direction, because while optimizing test sequences it might be quite important to have a valid sequence of method invocations in accordance with AFSM. Starting with in order set of input test sequences and applying MOGA using our fitness functions yield more useful test sequences.

3.6 Java based Tool for Research and Industry

While working on the research we have come up with a new tool which can be used as baseline for research in FSM based testing. The tool is now open source and freely available for subsequent researchers. This tool can build, save and load FSMs and run MOGA with custom fitness functions for generating optimized test sequences.

3.7 Summary

In this chapter we have presented our new proposed approach and have described different phases of the new approach. We have explained the process of parsing of OCL class contracts and their semantic analysis to generate the corresponding Abstract Finite State Machine. We have also explained the process of process of generating optimized test sequences from the generated AFSM. On the way we explain the coding scheme used, details of MOGA process on the coded chromosome and definition and evaluation of the fitness functions during the MOGA evolution process. At the end of the chapter we have listed some of the foreseen benefits achieved by the new proposed approach. At the end we include a discussion of the new tools developed using Java for implementation of the new approach.

CHAPTER 4

CASE STUDY AND EXPERIMENT

Generation of test sequences is a critical part of testing phase of software development life cycle. It is show by [1] that test sequences for unit testing of a class can be generated from OCL class specifications, that is by mapping class specifications (OCL class contracts) to the Class Model (specifically a Class in the class diagram).

Current test sequence generation process when applied to actual testing reveals some critical issues, these issues and our proposed solution is presented in this case study. CoinBox class is picked from a Drink Vending Machine's class diagram; this class is responsible for keeping record of number of available drinks and number of quarters entered by the customer. We used this class because it was used by the reference paper; it helps us to present a comparison. Two more classes Stack and Circle were tested.

4.1. Problems with previous Approach:

When we applied the previous approach to the generation of test cases from CoinBox class we observed the following problems. Current approach deviates from the actual OCL standards in terms of syntax and semantics. Due to the lack of conformance to standards of OCL the approach lacks the ability of automation. Due to state space exhaustive search the technique has inherent problems of the approach.

4.1.1. Deviation from standard OCL Syntax:

The example OCL code used by current approach [1] is not according to the Industry standard OCL syntax and hence none of OCL parsers used in the industry accepts this syntax e.g. the OCL example used is in the following format OCL Specs [11]:

```
Context CoinBox {
int curQtr, quantity, totalQtrs
boolean allowVend

inv    : int curQtr, quantity, totalQtrs >=0

:: CoinBox( )
post   : self.curQtr  = 0
         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

:: retQtrs( ):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 + curQtr@pre
:: addDrink(m: int):void   // add  m unit of drink in
                           // the machine
pre    : self.quantity == 0 and m > 0;
post   : self. quantity = quantity@pre + m
}
```

Fig.4.1. OCL Class Contract that does not comply with standard OCL

The above example was not according to the OCL standards syntax and after modification/adaptation we get the following OCL class contract that is acceptable according to the OCL 2.0 standard:

```
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.totalQtrs = 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 endif

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: slef.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
```

Fig.4.2. Actual Parse able OCL Class Contract.

As the used syntax deviates from the standard OCL in many aspects like e.g. [11]:

- Each statement in each pre and post condition is joined by a logical operator e.g. 'and', which is missing in the example.
- Standard OCL syntax does not allow the use of curly braces '{}' around the context declarations.
- All the OCL contexts (equivalent to Class) must be declared inside a package and endpackage statement.
- Each constraint in the Invariant declaration must be separated by 'and' instead of ';'.
- Writing just '::' operator while declaring a method signatures is not enough, it should be fully qualified with the context name being referred by the method.
- Each if must have an accompanying else in order to valid OCL statement.

4.1.1. Inherent Problems of Exhaustive Finite State Machine Exploration

Test sequences generated using the OCL class contract specifications using state-based technique suffer from their inherent problems including infeasible-paths and exponential number of generated test sequences. So we might get exponential number of test sequences which might also be of indefinite length. Use of these sequences might take exponential time for execution and even then we might not be sure if they cover even all the states of the object, along with that it is quite possible that a state is covered indefinite number of time e.g. if a state has a method loop (transition to itself with a method) or if a state is revisited again and again.

4.2. Application of our approach

Our approach works in following steps:

1. Generation of OCL Parse Tree: In our approach we take OCL class contract in the form of .ocl (a text file) and generate the parse tree for that passed file using an Industry standard OCL parser. At the moment we use Dresden OCL Parser [11]. This is a popular tool available both as standalone distribution and as an Eclipse integrated plug-in.
2. Semantic Analysis and Generation of AFSM: From the constructed parse tree, by semantic analysis of the tree and applying rules of the previous approach [1], an Abstract Finite State Machine is constructed.
3. Generation of Exhaustive search-based test sequences: By exhaustive search of the constructed AFSM, "Raw Test Sequences" are generated. These test sequences, although applicable for testing purpose, suffer from the state-based problems.
4. Optimization using MOGA: In-order to be processed by MOGA Optimization tool
    a. We have defined coding scheme for encoding solutions into genes and chromosomes as explained in detail, in section 3.2.1.
    b. We have plugged-in our custom mechanism to return random transitions (corresponding to the genes in our coding scheme) picked from the generated AFSM, used for mutation.
    c. Our custom devised, fitness functions, calculate the fitness values of each chromosome while selecting population for subsequent generation.

It is observed that the MOGA performance is highly dependent on the fitness functions used. The detailed process is shown in Fig.4.3.

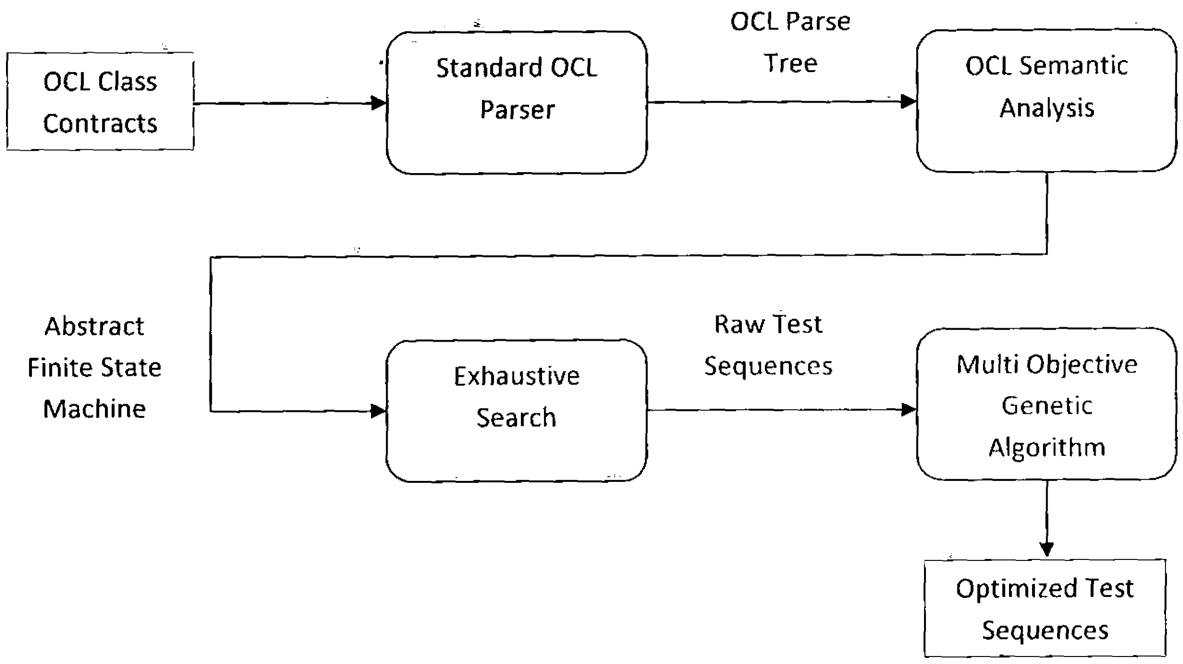


Fig.4.3. Automated MOGA optimized, test sequence generation process.

4.3. Mutation Analysis

We used mutation analysis for bench marking the performance of our approach. We used Mu Java[28] for seeding faults in the classes under test. It is important to be noted that proper selection of number of generations is problems 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.

Table 4.1. Mutation Analysis of CoinBox, Stack and Circle Classes

Class Under Test	Total Faults Seeded	Faults identified by Previous Approach	Faults Identified by New Approach
CoinBox	117	81	83
Stack	73	51	59
Circle	98	55	53

In this analysis predefined number of faults was seeded in the compiled class files. These faults were based on predefined mutation operators [28]. The experiment reveals that our approach either out-performs the previous approach or at least gives equal fault revealing efficiency.

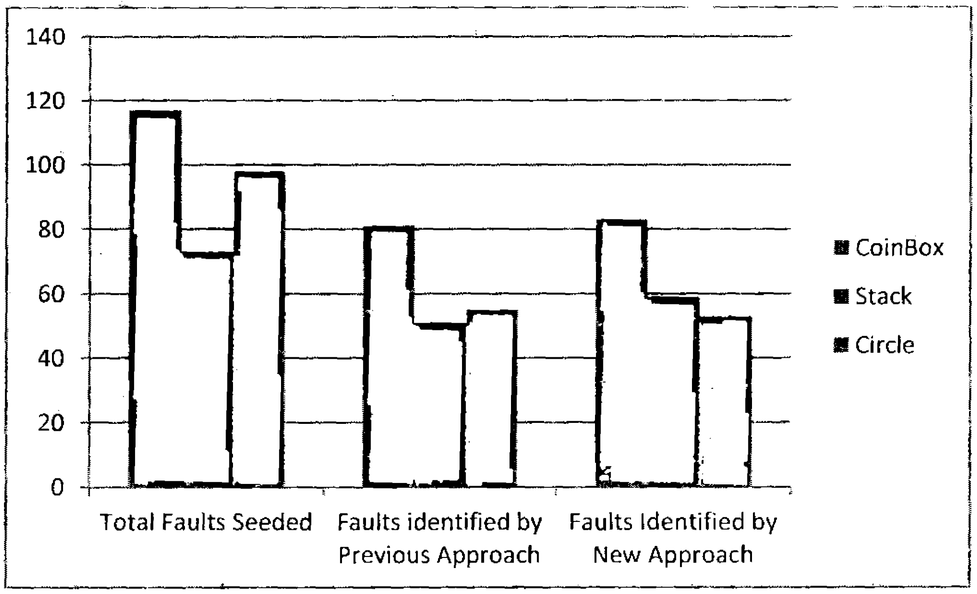


Fig.4.4. Comparison of fault identification efficiency of both the approaches.

One of the reasons of variations seems to be that current approach just follows transition tree coverage, which skips loops in the AFSM. Depending upon the nature of the class, loops might reveal more implementation errors, e.g. a method might give erroneous results on subsequent calls.

4.4. Advantages

Our approach improvements give following benefits to the research and industry community:

- Automation of the test sequence generation process, now test sequences can be generated directly from the OCL specifications of a class automatically.
- Test sequence generation even before the implementation of the software is ready.
- Helpful visual representation of generated Abstract Finite State Machine.
- Improved test sequences with specified length and number, we produced optimized test sequences of a certain length and having the maximum coverage of the states of the class.
- Fine Tuned fitness functions, fine tuned specifically for Test Sequence Optimization process.
- Less time and few resources required due to optimized test sequences, more reliable results because in exhaustive searching of class states we may never know how effective our testing is and when to stop.

4.5. Results and Discussion

Syntax of OCL used by current approach 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 import of class contract syntax used by current approach, revealed the syntax errors.

Our tool reads standard OCL class contracts and automatically generates the test sequences applying the rules used by 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.

As observed in the experimental case study exhaustive state space search generated 872 test sequences of maximum length 26 with redundant test sequence loops. Unnecessary effort needs to be spent on executing all these test sequences. Application of MOGA with population size 25, solution length 15 and over 1000 generations, yields 25 Test sequences of length 15 each; optimized for all transition coverage and ordered sequence paths. It was also observed that more generations give more diverse test sequences with higher fault revealing efficiency. Since we used a random population out of the search based sequences, it minimizes the chances of bad genes and evolution in negative direction

We did a mutation analysis of the class under test and found that MOGA based test sequence seem to give at least comparative defect revealing efficiency and may considerably outperform test sequence 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 of all optimization techniques, we are never expecting that we might have exact solution, but we get optimized solutions. MOGA being a subset of evolutionary algorithms starts with a possible set of solutions and then try to optimize the set of solutions generation after generation. Evolution as a 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 the nature if wrong genes get to the next generations then the individuals may suffer from defects. After generation of AFSM we can:

- Either generate a stochastic random population where each chromosome is a constituted out of a completely random set of genes
- Or get a random population out of the population of test sequences generated from state-based test sequence generation approach

Second option seems to give better results.

While specifying MOGA Fitness Functions for Test sequence Optimization we must take into account the sequence of Genes while calculating fitness values. Our approach gives improvement in terms of Automation of test sequence generation process. MOGA are quite effective while being used for test sequence optimization process but we recommend use of raw test sequences as initial population. MOGA optimized test sequences give optimized coverage within limited test sequence length and numbers.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1. Conclusion

The new proposed approach has improved the previous approach by conformance to industry standard syntax and automation from OCL to the actual test sequence generation. The new approach gives us benefits of optimization of 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 can save the time and resources spent on a part of testing process where selection of test sequences is done. Our approach gives improvement in terms of Automation of test sequence generation process.

Multi Objective Genetic Algorithms are quite effective while being used for test sequence optimization process and our 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 optimized test sequences give optimized coverage (maximum transitions coverage) within limited test sequence length and numbers.

We have also presented a Java based Open source tool, which can be used with any type of finite state machine while applying MOGAs; its scope is not just limited to the Class testing from OCL operation contract specifications. This tool 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.

5.2. Future Work

Some of the future work we want to do, as an improvement in our research work, is included in this section. We have automated the generation of test sequences from OCL operation contracts but complex OCL constrains and operators might need some additional attention and we would like to improve further the complex constraints handling functionally.

We have devised fitness functions very carefully but as there is always room for improvement, while using optimization techniques. So another future work might be improvement of the fitness functions to get better results in the generated test sequences.

Usage of variable length chromosomes seems to be a quite fantasizing phenomenon, but it is inherently complex and optimization of chromosome length to reduce length of test sequences needs investigation. An option is to add test sequence length optimization as an additional objective. In that way, some work is needed to be done to investigate the advantages and drawbacks of addition of length constraints.

We have tried our technique with experiments in the controlled laboratory environment. Another future course of research is to use our proposed approach to other industry applications and get feedback for improvement. We have already progressed in that way and our developed tool is available on the sourceforge.net for freely distribution under GNU license.

APPENDICES

Appendix A

Code for Test Sequence Optimization Fitness Functions

```
package pk.com.rsoft.ga.multiobjective;

import java.util.*;

import org.jgap.*;
import pk.com.rsoft.classcontractstestbed.testsequences.TestTransition;
import pk.com.rsoft.classcontractstestbed.util.graph.State;
import pk.com.rsoft.testsequenceoptimization.ga.*;

/**
* Fitness function for the test sequence problem.
*
* @author Rehan Farooq
*/
public class TestSequenceMOGAFitnessFunction extends BulkFitnessFunction {

private static final long serialVersionUID = 1L;

public void evaluate(Population a_subject) {
Iterator<IChromosome> it = a_subject.getChromosomes().iterator();
while (it.hasNext()) {
IChromosome a_chrom1 = it.next();
// Evaluate values to fill vector of multiobjectives with.
// ---------------------------------------------------
List<Double> l = new Vector<Double>();
l = CalculateFitness(a_chrom1);
((Chromosome) a_chrom1).setMultiObjectives(l);// Set fitness value
// for the Chrosome
}
}

public static Vector<Double> getVector(IChromosome a_chrom) {
List<?> moList = ((Chromosome) a_chrom).getMultiObjectives();
Vector<Double> retVector = new Vector<Double>();
retVector.add((Double) moList.get(0));
retVector.add((Double) moList.get(1));
return retVector;
}

@Override
```

```
public Object clone() {
return new TestSequenceMOGAFitnessFunction();
}

public Vector<Double> CalculateFitness(IChromosome a_Chromosome) {
Vector<Double> v = new Vector<Double>();
v.add(getFitnessByTransitionOrder(a_Chromosome));// Fitness by validity
// is in
// location indexed 0
v.add(calculateFitnessByCoverage(a_Chromosome));// Fitness by Coverage
// is in location
// indexed 1
return v;
}

private Double calculateFitnessByCoverage(IChromosome a_Chromosome) {
double retVal = 0;
State from;
State to;

Gene[] genes = a_Chromosome.getGenes();
if (((TestTransition) genes[1].getAllele()).getFromState()
.isStartState()) {
retVal += 20;
}
int occuranceCount = 0;
for (State state : AFSMHolder.getFSM().getStatesList()) {
occuranceCount = 0;

for (int index = 0; index < genes.length; index++) {
from = ((TestTransition) genes[index].getAllele())
.getFromState();
to = ((TestTransition) genes[index].getAllele()).getToState();
if (state.isSameAs(from)) {
occuranceCount++;
}
if (state.isSameAs(to)) {
occuranceCount++;
}
}
if (occuranceCount == 1 || occuranceCount == 2) {
retVal += 3 * occuranceCount;
} else if (occuranceCount == 0 || occuranceCount > 2) {
retVal -= 10;
}
}

return new Double(retVal);
}

private Double getFitnessByTransitionOrder(IChromosome a_Chromosom) {

double retVal = 0;
Gene[] genes = a_Chromosom.getGenes();

if (((TestTransition) genes[0].getAllele()).getFromState()
.isStartState()) {
retVal += 20;
}
State next = ((TestTransition) genes[0].getAllele()).getToState();
for (int index = 1; index < genes.length; index++) {
if (next.isReachable(((TestTransition) genes[index].getAllele())
.getToState())) {
retVal += 5;
} else {
retVal -= 5;
}
next = ((TestTransition) genes[index].getAllele()).getToState();
}
return new Double(retVal);

}
}
```

Appendix B

Java Code for Context Class, Root Class of All OCL Elements

```
package pk.com.rsoft.classcontractstestbed.classcontract;

import java.io.Serializable;
import javax.swing.tree.TreeNode;
import java.util.ArrayList;
import java.util.HashSet;
import javax.swing.tree.DefaultMutableTreeNode;

import pk.com.rsoft.classcontractstestbed.util.graph.ClassVariable;
import pk.com.rsoft.classcontractstestbed.util.inequality.InEqualitySimplified;
import pk.com.rsoft.classcontractstestbed.util.parser.CTStringParser;

/**
* @author Rehan Farooq
*/
public class CTContext implements Serializable {
private ArrayList<CTAttribute> lstAttributes;
private ArrayList<CTOperation> lstOperations;
private CTInvarient theInvarient;
private String strCTContextName;
private TreeNode theContextNode;

public CTContext(TreeNode theNode) {
strCTContextName = "";
lstAttributes = new ArrayList<CTAttribute>();
lstOperations = new ArrayList<CTOperation>();
theContextNode = theNode;
parseContextNode(theNode);
}

public void addAttribute(TreeNode attNode) {
CTAttribute atr = new CTAttribute(attNode);
this.addAttribute(atr);
}

public void addOperation(TreeNode optNode) {
CTOperation tempOp = new CTOperation(optNode);
if (tempOp.getCTOperationName().equals(this.getCTContextName())) {
tempOp.setConstructor(true);
}
getLstOperations().add(tempOp);

}
```

```
public void addAttribute(CTAttribute anAttrib) {
if (!isDuplicateAttribute(anAttrib)) {
lstAttributes.add(anAttrib);
}
}
public void addOperation(CTOperation anOpp) {
getLstOperations().add(anOpp);
}
public void addInvarient(CTInvarient anInv) {
this.setInvarient(anInv);
}
public void addInvarient(TreeNode invNode) {
this.setInvarient(new CTInvarient(invNode));
}
private void parseContextNode(TreeNode ctNode) {
if (ctNode.isLeaf()) {
this.setCTContextName(ctNode.toString().trim());
this.setCTContextName(CTStringParser
.extractNameFromQuotes(getCTContextName()));
} else {
DefaultMutableTreeNode tm = (DefaultMutableTreeNode) ctNode;
parseContextNode(tm.getChildAt(0));
}
}
/**
* @return the lstAttributes
*/
public ArrayList<CTAttribute> getLstAttributes() {
return lstAttributes;
}
/**
* @return the lstOperations
*/
public ArrayList<CTOperation> getLstOperations() {
return lstOperations;
}
/**
* @return the strCTContextName
*/
public String getCTContextName() {
return strCTContextName;
}

/**
* @param strCTContextName
```

```
*	the strCTContextName to set
*/
public void setCTContextName(String strCTContextName) {
this.strCTContextName = strCTContextName;
}

/**
* @return the theInvarient
*/
public CTInvarient getInvarient() {
return theInvarient;
}

/**
* @param theInvarient
*	the theInvarient to set
*/
public void setInvarient(CTInvarient theInvarient) {
this.theInvarient = theInvarient;
}

public String getVaiableNames() {
/*
* Very Very Important! This code extracts the varibale and methods from
* the OCL but at the moment there is a constraint on the varibale
* declartion that is only those variables are picked which are declared
* in the OCL Init statements (in future there can be a possibility of
* infering from the pre and post conditions but at the moment it is
* implemented that way)
*/
StringBuilder strRetVal = new StringBuilder("");
if (this.lstAttributes != null) {
for (CTAttribute at : this.lstAttributes) {
strRetVal.append(at.getType()).append(" ").append(at.getName())
.append(",");
}
} else {
System.out.println("Variables Names list empty");
}
return strRetVal.toString();
}

public String getOperationNames() {
StringBuilder strRetVal = new StringBuilder("");
if (this.lstOperations != null) {
for (CTOperation op : this.lstOperations)
```

```
strRetVal.append(op.getCTOperationName()).append(",");

}
return strRetVal.toString();
}

public ArrayList<CTAttribute> getStateVariables() {
ArrayList<CTAttribute> lst = new ArrayList<CTAttribute>();
for (CTAttribute atr : this.lstAttributes) {
for (CTOperation op : this.lstOperations) {
if (op.isPreCondtionVariable(atr.getName())) {
boolean duplicate = false;
for (CTAttribute at : lst) {
if (at.getName().equals(atr.getName())) {
duplicate = true;
}
}
if (!duplicate) {

lst.add(atr);
}
}
}
}

return lst;
}

public String getStateVariableNames() {
HashSet<String> st = new HashSet<String>();
for (CTAttribute atr : this.lstAttributes) {
for (CTOperation op : this.lstOperations) {
if (op.isPreCondtionVariable(atr.getName())) {
st.add(atr.getName());
}
}
}

return st.toString();
}

public ArrayList<String> getAllValuesStrings(String strVarName) {

ArrayList<String> retVals = new ArrayList<String>();
for (CTOperation op : this.lstOperations) {
for (String val : op.getVarValues(strVarName, ConstraintType.PRE)) {
```

```
if (!Containts(retVals, val)) {
retVals.add(val.trim());
}
}
for (String val : op.getVarValues(strVarName, ConstraintType.POST)) {
if (!Containts(retVals, val)) {
retVals.add(val.trim());
}
}
}
for (CTConstraint ctx : this.theInvarient.lstConstraints) {
if (ctx.getVariableName().equals(strVarName)
&& !ctx.getVariableValue().trim().equals("")) {
if (!Containts(retVals, strVarName)) {

retVals.add("Inv " + ctx.getVariableValue().trim());
}
}
}
return retVals;
}

private boolean Containts(ArrayList<String> lst, String str) {
for (String s : lst) {
if (s.equals(str)) {
return true;
}
}
return false;
}

private boolean Containts(String strVal, ArrayList<InEqualitySimplified> lst) {
for (InEqualitySimplified s : lst) {
if (s.getVariable().getValue().trim().equals(strVal.trim())) {
return true;
}
}
return false;
}

public String getInvarientsDesc() {
StringBuilder strBld = new StringBuilder();
for (CTConstraint ct : this.theInvarient.getConstrantsList()) {
strBld.append(ct.getVariableName()).append(" ")
.append(ct.getVariableValue()).append(", ");
}
```

```
return strBld.toString();
}

public void ClearLists() {
this.lstAttributes.clear();
this.lstOperations.clear();
this.theInvarient.lstConstraints.clear();
}

private boolean isDuplicateAttribute(CTAttribute atrib) {
for (CTAttribute at : lstAttributes) {
if (at.getName().equals(atrib.getName())) {
return true;
}
}
return false;
}

/*
* This method returns the state variables with simplified set of
* possible values These values should not include values having '@pre' etc
* but should have the refined possible set of values
*/
public ArrayList<ClassVariable> getStateVar() {
// StateVaiable List to return
ArrayList<ClassVariable> lstStateVars = new ArrayList<ClassVariable>();
// Current variables of interest
ArrayList<CTAttribute> lstAttrib = getStateVariables();

for (CTAttribute atr : lstAttrib)// for each attribute
{
// Create new state variable to return
ClassVariable stVar = new ClassVariable(atr.getName(),
atr.getInitVal(), atr.getType());

// Get All the values attached with this attribute
ArrayList<String> lstVals = getAllValuesStrings(atr.getName());
for (String str : lstVals)// for each attribute value
{
if (str.contains("@pre"))// if the value contains @pre tag try to asses possibie output
value(s)
{
String temp = "";
for (CTOperation op : this.lstOperations) {
if (op.isPreCondtionVariable(stVar.getName())) {
temp = str;
if (!Containts(temp, stVar.getValues()))// Add if to
// values if
// not
// already
// added
{
InEqualitySimplified tempInq = new InEqualitySimplified(
atr.getName() + " " + temp,
getVariableType(atr.getName()));
stVar.addValue(tempInq);
}
}
}
} else // No @Pre tag
{
if (!Containts(str, stVar.getValues())) // Add it to values
// if not already
// added
{
InEqualitySimplified tempInq = new InEqualitySimplified(
atr.getName() + " " + str,
getVariableType(atr.getName()));
stVar.addValue(tempInq);
}
}
}

lstStateVars.add(stVar);
}
return lstStateVars;
}

public CTVariableType getVariableType(String strVarName) {
for (CTAttribute att : this.lstAttributes) {
if (att.getName().trim().equals(strVarName.trim())) {
return att.getCTType();
}
}
return CTVariableType.OTHER;
}

public TreeNode getContextNode() {
return this.theContextNode;
}
}
```

Appendix C

Java Code of Abstract Finite State Machine

```
/*

* This is the class representing an Abstract State Machine

* it is responsible for creating, maintaining and running the AFSM Model

*/

package pk.com.rsoft.classcontractstestbed.util.graph;


import java.awt.Graphics;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

import java.util.ArrayList;


import javax.swing.JOptionPane;


import pk.com.rsoft.classcontractstestbed.ClassVarDialog;

import pk.com.rsoft.classcontractstestbed.classcontract.*;

import pk.com.rsoft.classcontractstestbed.testsequences.TestTransition;

import pk.com.rsoft.classcontractstestbed.testsequences.TestSequence;

import pk.com.rsoft.classcontractstestbed.util.graphics.Point;

import pk.com.rsoft.classcontractstestbed.util.graphics.Shape;

import pk.com.rsoft.classcontractstestbed.util.inequality.InEqualitySimplified;

import pk.com.rsoft.classcontractstestbed.util.inequality.InequalityOperatorType;

import pk.com.rsoft.classcontractstestbed.util.inequality.InequalitySolver;
import pk.com.rsoft.classcontractstestbed.util.inequality.InequationSolver;
import pk.com.rsoft.classcontractstestbed.util.parser.OperatorType;

/**
* @author Rehan Farooq
*/
public class AbstractFSM implements Shape, Serializable {
private static final long serialVersionUID = 1L;
private ArrayList<State> lstStates;// The Array List containing all the
// states of the AFSM
static int intVal = 0;
// IMPORTANT: CLASS Variables are in the context of a Class and STATE
// Variables are in the context of AFSM State
private StringBuilder strLog = new StringBuilder();
private ArrayList<ClassVariable> lstStateVars;// The Array List containing
private CTContext ctx;// The Parsed Context for which AFSM is being built
ArrayList<TestSequence> testSequences;
final int START_X = 200;
final int X_INCREMENT = 150;
final int START_Y = 200;
final int Y_INCREMENT = 150;

/**
* This Empty parameters constructor just initializes the state of AFSM with
* Empty values! Caution: AFSM might not be useful after just this
* initialization!
*/
```

```
public AbstractFSM() {
this(null);// class the other constructor the DRY principal!
}

public AbstractFSM(CTContext ct) {
this.setStatesList(new ArrayList<State>());
this.lstStateVars = new ArrayList<ClassVariable>();
if (ct == null) {
return;// if ct CTContext is null no need to go further
}
this.setCtx(ct);// Record the this AFSM was built using this ct
// ****************************//
// Start of State Variables setup//
// ****************************//
initializeStateVariables();// Initialization of statate

System.out.println("\nNumber of variables -->" + lstStateVars.size());
System.out.println("State Variables as initialized :");
System.out.println(this.lstStateVars);

ArrayList<ClassVariable> lst = simp(ct.getLstOperations(),
this.lstStateVars);

System.out.println("\nList of states is Simplified:");// That is
// constraints
// having @pre
// are
// simplified
System.out.println(lst);

lst = removeExtraEqual(lst);
System.out.println("\nAfter Removing unwanted EQUAL!");
System.out.println(lst);

lst = validateVariables(lst);
System.out.println("\nAfter Validation!");// That is removing unwanted
// and invalid values
System.out.println(lst);

this.lstStateVars = lst;

lst = convertAllToAtomic(lst);
System.out.println("\nAutomic vals the final Class Variable List:");
System.out.println(lst);
// ***************************//
// End of State Variables setup//
// ***************************//

logAction(lst.toString());// log the value of state variabels after
// setup
lstStateVars = lst;
}

public void EditClassVariables() {
```

```
ClassVarDialog dlg = new ClassVarDialog("Class Variables", true,
this.lstStateVars);
dlg.setVisible(true);
}

public boolean buildAFSM() {
buildInitialStates();// Now buildup states

if (this.lstStateVars.size() < 1) {
JOptionPane
.showMessageDialog(null,
"No Variables of Interest found, this problems is not solveable!");
return false;
}
this.lstStates = processAllStates(lstStates);
adjustStatePositions();
return true;
}

private ArrayList<ClassVariable> simp(ArrayList<CTOperation> ctOps,
ArrayList<ClassVariable> theList) {
ArrayList<ClassVariable> lstStVars = new ArrayList<ClassVariable>();
for (ClassVariable var : theList) {
lstStVars.add(simplyfy(var, ctOps));
}
return lstStVars;
}
```

```
private ClassVariable simplyfy(ClassVariable aVar,
ArrayList<CTOperation> ctOps) {
ClassVariable stVar = null;
if (aVar.getType() != CTVariableType.INTEGER
&& aVar.getType() != CTVariableType.REAL) {
stVar = aVar;
} else {
stVar = new ClassVariable(aVar.getName(), aVar.getType());
ArrayList<InEqualitySimplified> temInqs = new ArrayList<InEqualitySimplified>();
for (CTOperation op : ctOps) {
ClassVariable tempVar = getOutputValue(aVar, this.getContext(),
op);
for (InEqualitySimplified inq : tempVar.getValues()) {
if (!inq.getVariable().getValue().toUpperCase()
.contains("@PRE")) {
inq.getVariable().setName(aVar.getName());
if (!InqContainedInList(inq, temInqs)) {

temInqs.add(inq);
}
}
}
}
stVar.setVarValues(temInqs);

}
```

```
return stVar;
}

private void buildInitialStates() {
CTContext theContext = this.getContext();
lstStates.clear();
// Get the List of All Class Contract Operations
ArrayList<CTOperation> lstOpts = getContext().getLstOperations();
// Get the List of All state variables
// For All Operations in the Class Contract try to construct Absrtact
// inital States
for (CTOperation opt : lstOpts) {
if (opt.isConstructor()) {
State state = createSate(theContext, opt, true);
lstStates.add(state);
}
}
}
/**
* @param theContext
* @param opt
*/
private State createSate(CTContext theContext, CTOperation opt,
boolean isStartState) {
// This is a Constructor create a new Initial abstract state
State retState = new State(isStartState);
for (ClassVariable clsVar : this.lstStateVars) {
```

```
if (opt.isPostCondtionVariable(clsVar.getName())) {
// It is in the post conditions so build it's output value
String theVal = this.getPostConditionValue(clsVar.getName(),
opt.getPostConditions());
if (theVal != null) {
retState.addStateVariable(new InEqualitySimplified(theVal,
clsVar.getType()));
}
} else {//
// It is not in the post condition so it's value from
// defaults will be picked
InEqualitySimplified defaultval = InequalitySolver
.getDefaultValue(clsVar, theContext);
retState.addStateVariable(defaultval);// add this value to state
// variables of new
// state
}

}// End for(ClassVariable v : this.lstStateVars)
retState.setUnprocessed();
return retState;
}
private ArrayList<State> getNextStates(State st, ArrayList<State> lstArray) {
ArrayList<CTOperation> lstOps = getContext().getLstOperations();
ArrayList<State> retList = new ArrayList<State>();
for (CTOperation ops : lstOps) {
if (isOperationExecutable(st, ops)) {
```

```
System.out.println("Operation being processed "
+ ops.getCTOperationName());
State tempSt = getNextState(st, ops, lstArray);
if (tempSt != null) {
st.addTransition(new Transition(st, tempSt, ops));
retList.add(tempSt);
} else {
System.out.println("No transition added ");
}
}
}
return retList;
}
private State getNextState(State st, CTOperation ops,
ArrayList<State> lstArray) {
State retState;
if (isOperationExecutable(st, ops)) {
ArrayList<InEqualitySimplified> nextVals = getNextValues(
st.getCurrentValues(), ops, true);
State stat = findInList(lstArray, nextVals);
if (stat == null) {
retState = new State(false);
System.out.println("Current State count --> " + intVal);
retState.setCurrentValues(nextVals);
} else {
retState = stat;
}
```

```
return retState;

}


return null;

}

private ArrayList<InEqualitySimplified> getNextValues(

ArrayList<InEqualitySimplified> preValues, CTOperation ops,

boolean simplifyIt) {

ArrayList<InEqualitySimplified> retInqs = new ArrayList<InEqualitySimplified>();

for (InEqualitySimplified theIneq : preValues)// for each value of

// variables in the

// preValues

{

if (ops.isPostCondtionVariable(theIneq.getVariableName().trim()))

{

String strVal = ops.getVarValue(theIneq, preValues,

                                 ConstraintType.POST);// Get return constraint
                                 value from  the operation

if (simplifyIt && strVal.toUpperCase().trim().contains("@PRE"))// if

{

if (!OperatorType.isArithmeticExpression(strVal)) {

retInqs.add(theIneq);

} else {

InEqualitySimplified newVal = new InEqualitySimplified(

theIneq.getVariableName() + strVal,

theIneq.getVariableType());

newVal = InequalitySolver.simplify(newVal, ops,

getContext(), preValues, lstStateVars);
newVal.getVariable().setName(theIneq.getVariableName());
retInqs.add(newVal);
}
} else// Just return the InEqualitySimplified without
// simplification if any
{
InEqualitySimplified newVal = new InEqualitySimplified(
theIneq.getVariableName() + strVal,
theIneq.getVariableType());
retInqs.add(newVal);
}

} else// if it is not a post condition variable
{
retInqs.add(theIneq);// just add it as unchanged value
}
}
return retInqs;
}
private boolean isOperationExecutable(State s, CTOperation op) {
if (op.isConstructor()) {
return false;
}
ArrayList<InEqualitySimplified> lstCurrentVals = s.getCurrentValues();// States
ArrayList<InEqualitySimplified> preConditions = InequalitySolver
.getPreValues(op, getContext());// Pre condition values of the
// variables
```

```
boolean retVal = true;
for (InEqualitySimplified varVal : lstCurrentVals)// for each current  value
{
for (InEqualitySimplified preVal : preConditions)// for each pre  condition
{
if (varVal.getVariableName().trim()
.equals(preVal.getVariableName().trim()))
{
if (!InequationSolver.isTheSame(varVal, preVal)) {
String strPreOperator = InequalityOperatorType
.toString(preVal.getType());
if (strPreOperator.trim().equals("="))// if operator is
{
strPreOperator += "=";
}
strPreOperator = " " + strPreOperator + " "; // just add
Object obj1 = InequationSolver
.evaluate(varVal.getVariable().getValue().toLowerCase()   + strPreOperator
+ preVal.getVariable().getValue().toLowerCase());
if (obj1 != null) {
retVal = retVal && Boolean.valueOf(obj1.toString());
}
if (strPreOperator.trim().equals("==")) {
String strPostOperator = nequalityOperatorType
.toString(varVal.getType());
if (strPostOperator.equals("=")) {
strPostOperator += "=";
}
obj1 = InequationSolver.evaluate(preVal.getVariable().getValue().toLowerCase() +
strPostOperator+ varVal.getVariable().getValue().toLowerCase());
if (obj1 != null) {
retVal = retVal && Boolean.valueOf(obj1.toString());
}
}}
}}
}
if (retVal == true)
System.out.println(op.getCTOperationName() + " is executable!");
else
System.out.println(op.getCTOperationName() + " is not executable!");
return retVal;
}
private void logAction(String strAction) {
strLog.append(strAction).append("\n");
}
public void clearLog() {
setLog(new StringBuilder().toString());
}
/**
* @return the strLog
*/
public String getLog() {
return strLog.toString();
}
/**
```

```
* @param strLog
* the strLog to set
*/
public void setLog(String strLog) {
this.strLog = new StringBuilder(strLog);
}

public ClassVariable getOutputValue(ClassVariable currentVar,
CTContext context, CTOperation op) {
ClassVariable retVal = currentVar;
if (!op.isPostCondtionVariable(currentVar.getName()))// if that veriable
{
return retVal;
} else {
ClassVariable var = simplifyIt(currentVar, context, op);
return var;
}
}

private ClassVariable simplifyIt(ClassVariable currentVar,
CTContext context, CTOperation op) {
ClassVariable stVar = new ClassVariable(currentVar.getName(),
currentVar.getValueAt(0).toString(), currentVar.getType());
// No this is the variable in the post condition of current operation we
// need to
// if we need processing to asses @pre key words
for (InEqualitySimplified inq : currentVar.getValues()) {
```

```
if (inq.getVariable().getValue().toUpperCase().contains("@PRE")) {
stVar.addValue(InequalitySolver.simplify(inq, op, context));
} else {
stVar.addValue(inq);
}
}
return stVar;
}
private void initializeStateVariables() {
if (this.getContext() == null) {
return;
}
ArrayList<ClassVariable> lstVars = getContext().getStateVar();
InEqualitySimplified trueVal = new InEqualitySimplified(
"trueVal = TRUE", CTVariableType.BOOLEAN);
InEqualitySimplified falseVal = new InEqualitySimplified(
"falseVal = FALSE", CTVariableType.BOOLEAN);

ArrayList<InEqualitySimplified> lstInq = new ArrayList<InEqualitySimplified>();
lstInq.add(trueVal);
lstInq.add(falseVal);
for (ClassVariable var : lstVars) {
if (var.getType() == CTVariableType.BOOLEAN) {
trueVal.getVariable().setName(var.getName());
falseVal.getVariable().setName(var.getName());
var.setVarValues(lstInq);
}
```

```
}
this.lstStateVars = lstVars;
}

public boolean containsInPreCondition(CTOperation opt, ClassVariable var) {
for (CTPreCondition pre : opt.getPreConditions()) {
if (!pre.getVarVals(var.getName()).isEmpty()) {
return true;
}
}
return false;
}

public boolean containInPostCondition(CTOperation opt, ClassVariable var) {
for (CTPostCondition post : opt.getPostConditions()) {
if (post.isInCondition(var.getName())) {
return true;
}
}
return false;
}

private boolean InqContainedInList(InEqualitySimplified inq,
ArrayList<InEqualitySimplified> lst) {
for (InEqualitySimplified simpInq : lst) {
if (InequationSolver.isTheSame(inq, simpInq)) {
return true;
```

```
}
}
return false;
}


private String RemoveEqualfromVal(String val) {
String retVal = val;
if (val.contains("EQUAL")) {
retVal = val.substring(val.lastIndexOf("EQUAL") + "EQUAL".length());
}
return retVal;
}
private ArrayList<InEqualitySimplified> convertToAtomic(
ArrayList<InEqualitySimplified> lst) {
ArrayList<InEqualitySimplified> retList = new ArrayList<InEqualitySimplified>();
for (InEqualitySimplified inq : lst) {
if (InequalityOperatorType.isComposit(inq.getType())) {
ArrayList<InEqualitySimplified> tempList = InequationSolver
.split(inq);
for (InEqualitySimplified simp : tempList) {
if (!InqContainedInList(simp, retList)) {
retList.add(simp);
}
}
} else {
if (!InqContainedInList(inq, retList)) {
retList.add(inq);
```

```
}}
}
return retList;
}
private String getPostConditionValue(String strName,
ArrayList<CTPostCondition> postCons) {
String retVal = null;
for (CTPostCondition post : postCons) {
retVal = post.getVarVal(strName).trim();
if (retVal != null && retVal != "") {
return strName + retVal.replace(",", " ").trim();
}
}
return retVal;
}

private ArrayList<State> processAllStates(ArrayList<State> inputlst) {
ArrayList<State> retList = new ArrayList<State>();
retList.addAll(inputlst);

while (hasUnProcessedStates(retList)) {
for (int index = 0; index < retList.size(); index++) {
State stTemp = retList.get(index);
if (!stTemp.isProcessed()) {
ArrayList<State> stNextStatesList = getNextStates(stTemp,
retList);
for (State state : stNextStatesList) {
```

```
if (!isStateInTheList(state, retList)) {
retList.add(state);
}}
stTemp.setProcessed();
}}
}
return retList;
}
private boolean hasUnProcessedStates(ArrayList<State> lst) {
for (State st : lst) {
if (!st.isProcessed()) {
return true;
}
}
return false;
}
private boolean isStateInTheList(State state, ArrayList<State> stateList) {
for (State st : stateList) {
if (st.isSameAs(state)) {
return true;
}
}
return false;
}

/**
* @param lstStates
*         the lstStates to set
*/
public void setStatesList(ArrayList<State> lstStates) {
this.lstStates = lstStates;
}
/**
* @return the lstStates
*/
public ArrayList<State> getStatesList() {
return lstStates;
}
@Override
public String toString() {
StringBuilder retStr = new StringBuilder();
retStr.append("[");
for (State s : this.lstStates) {
retStr.append(s.toString());
retStr.append(",");
}
retStr.append("]");
return retStr.toString();
}

private ArrayList<ClassVariable> removeExtraEqual(
ArrayList<ClassVariable> lst) {
for (ClassVariable var : lst) {
if (var.getType() == CTVariableType.INTEGER
|| var.getType() == CTVariableType.REAL)// form all integer
// and real
// variables
{
ArrayList<InEqualitySimplified> temList = new ArrayList<InEqualitySimplified>();
for (InEqualitySimplified inSmp : var.getValues()) {
if (inSmp.getVariable().getValue().contains("EQUAL"))// if
{
inSmp.getVariable().setValue(
RemoveEqualfromVal(inSmp.getVariable()
.getValue()));
}
if (!InqContainedInList(inSmp, temList)) {
temList.add(inSmp);
   }
}
var.setVarValues(temList);


}
}
ArrayList<ClassVariable> retList = lst;
return retList;
}


private ArrayList<ClassVariable> validateVariables(
ArrayList<ClassVariable> lst) {
for (ClassVariable avar : lst)// This loop checks for the validity of
```

```
{
ArrayList<InEqualitySimplified> lstInverients = InequalitySolver
.getInvarients(avar, getContext());

ArrayList<InEqualitySimplified> temInqs = new ArrayList<InEqualitySimplified>();
temInqs.addAll(avar.getValues());
for (InEqualitySimplified temInq : temInqs) {
if (!InequalitySolver.isValid(temInq, lstInverients)) {
avar.getValues().remove(temInq);
}}
}
ArrayList<ClassVariable> retLst = lst;
return retLst;
}

private ArrayList<ClassVariable> convertAllToAtomic(
ArrayList<ClassVariable> lst) {
for (ClassVariable vars : lst)// This for loop splits the composite
// inequalities (having >= and <=) to
// atomic
{
ArrayList<InEqualitySimplified> temp = convertToAtomic(vars
.getValues());
vars.setVarValues(temp);
}
ArrayList<ClassVariable> retList = lst;
return retList;
```

```
}

private State findInList(ArrayList<State> lst,
ArrayList<InEqualitySimplified> stateVals) {
for (State st : lst) {
if (isStateSame(st, stateVals)) {
return st;
}
}
return null;
}

private boolean isStateSame(State s, ArrayList<InEqualitySimplified> list) {
boolean retVal = true;
for (InEqualitySimplified inq : s.getCurrentValues()) {
for (InEqualitySimplified inqFromList : list) {
if (inq.getVariableName().trim()
.equals(inqFromList.getVariableName().trim()))// have
// same
// variable
// names
{
if (!InequationSolver.isTheSame(inq, inqFromList))// are
// they the same
{
// If not the same then return false here
retVal = false;
```

```
}}
}}
return retVal;
}

private void adjustStatePositions() {
int count = 1;
int xAxis = 100;
int yAxis = 60;
int xDistance = 200;
int yDistance = 0;
for (State st : lstStates) {
st.setX((count) * xDistance + xAxis);
st.setY(yAxis + count++ * yDistance);
}}

public void generateTestSequences() {
testSequences = new ArrayList<TestSequence>();
ArrayList<TestSequence> copyList = new ArrayList<TestSequence>();
if (lstStates.size() > 0) {
int i = 0;
while (i++ < 25) {
if (testSequences.size() < 1)// If this is the first test
// sequence
{
State firstState = lstStates.get(0);
for (Transition trans : firstState.getArNextStates()) {
```

```
TestSequence tempSequence = new TestSequence();
tempSequence.addToSequence(new TestTransition(trans
.getMethod(), trans.getFromState(), trans
.getToState()));
testSequences.add(tempSequence);
}

} else // we have already got our first test sequence
{
copyList.addAll(testSequences);
for (TestSequence seq : copyList)// for each test sequence
// in the test sequences
{
TestTransition theCall = seq.getSequence().get(
seq.getSequence().size() - 1);// get the last
// method call
Transition nextTrans = theCall.getToState()
.getNextState(0);
seq.getSequence().add(
new TestTransition(nextTrans.getMethod(),
nextTrans.getFromState(), nextTrans
.getToState()));// just add this
// call the
// sequence

if (theCall.getToState().getTransitionCount() > 1) {
for (int index = 1; index < theCall.getToState()
.getTransitionCount() - 1; index++) {
ArrayList<TestTransition> newSeq = new ArrayList<TestTransition>();
newSeq.addAll(seq.getSequence());

nextTrans = theCall.getToState().getNextState(
index);
newSeq.add(new TestTransition(nextTrans
.getMethod(), nextTrans.getFromState(),
nextTrans.getToState()));
testSequences.add(new TestSequence(newSeq));
}}
}
copyList.clear();
}}
}}

public ArrayList<TestSequence> getNTestSequences(int noSequences)
throws Exception {
if (noSequences > testSequences.size()) {
throw new Exception(
"Desired number of test sequnces is higher than the available sequences. available No : "
+ testSequences.size()
+ " desired No :"
+ noSequences);
}
return new ArrayList<TestSequence>(
testSequences.subList(0, noSequences));
```

```
}

public ArrayList<TestSequence> getGeneratedSequences() {
return testSequences;
}
@Override
public void draw(Graphics g) {
// TODO Auto-generated method stub
adjustStateCoordinates(20, 20, true);
for (State state : this.lstStates) {
state.draw(g);
}}
@Override
public void Move(int newX, int newY, Graphics g) {
// TODO Auto-generated method stub
//Not Implemented!
}
public static AbstractFSM fromFile(String fileNamewithPath) {
FileInputStream fin = null;
ObjectInputStream objIn = null;
try {
fin = new FileInputStream(fileNamewithPath);
objIn = new ObjectInputStream(fin);
return (AbstractFSM) objIn.readObject();
} catch (Exception ex) {
JOptionPane.showMessageDialog(null, ex);
}
```

```
return null;
}
public static boolean toFile(AbstractFSM fsm, String fnameWithPath) {
FileOutputStream fout = null;
ObjectOutputStream oout = null;
try {
fout = new FileOutputStream(fnameWithPath);
oout = new ObjectOutputStream(fout);
oout.writeObject(fsm);
} catch (Exception ex) {
JOptionPane.showMessageDialog(null, ex.toString());
return false;
}
return true;
}

public CTContext getContext() {
return ctx;
}

private void setCtx(CTContext ctx) {
this.ctx = ctx;
}

public ArrayList<String> getClassVariableNames() {
ArrayList<String> retList = new ArrayList<String>();
for (ClassVariable var : this.lstStateVars) {
```

```
retList.add(var.getName());
}
return retList;
}
public void adjustStateCoordinates(int x, int y, boolean evenODD) {
int i = 1;
for (State st : this.lstStates) {
if (evenODD) {
if (i++ % 2 == 0) {
st.setCenter(new Point(x, y));
}

} else {
st.setCenter(new Point(x, y));
}}
}
public void setLocationOnScreen(int x, int y, boolean xConstant) {
int fixIncrement = 20;
int intStateCount = 1;
for (State st : this.lstStates) {
if (!xConstant) {
st.setCenter(new Point(x + fixIncrement * (intStateCount), y
+ fixIncrement * (intStateCount)));
} else {
st.setCenter(new Point(x, y + fixIncrement * (intStateCount)));
}
intStateCount++;}}}
```

Appendix D

Java Code for Test Transition

```
package pk.com.rsoft.classcontractstestbed.testsequences;

import java.io.Serializable;

import java.util.ArrayList;

import jmetal.core.Variable;

import org.jgap.InvalidConfigurationException;

import pk.com.rsoft.classcontractstestbed.classcontract.CTConstraint;

import pk.com.rsoft.classcontractstestbed.classcontract.CTMethodParameter;

import pk.com.rsoft.classcontractstestbed.classcontract.CTOperation;

import pk.com.rsoft.classcontractstestbed.classcontract.ConstraintType;

import pk.com.rsoft.classcontractstestbed.util.graph.ConsType;

import pk.com.rsoft.classcontractstestbed.util.graph.State;

import pk.com.rsoft.classcontractstestbed.util.graph.Transition;

import pk.com.rsoft.classcontractstestbed.util.inequality.InEqualitySimplified;

import pk.com.rsoft.classcontractstestbed.util.inequality.InequalitySolver;

import pk.com.rsoft.classcontractstestbed.util.parser.NumberPorcessor;

import pk.com.rsoft.testsequenceoptimization.ga.AFSMHolder;


/**
 * @author Rehan Farooq
 */
public class TestTransition implements Serializable {
        private static final long serialVersionUID = 1L;
        private final int FIX_NUMBER = 5;
        private State preState ;
        private State postState;
        CTOperation opt;

        public TestTransition(CTOperation op, State preState, State postState) {

                this.setFromState(preState);

                this.setToState(postState);

                this.opt = op;

        }


        public String toBooleanString() throws InvalidConfigurationException

    {

                String retVal =
getBoolStateString(getFromState())+getMethodBoolString()+getBoolStateString(getToSt
ate());

                return retVal;

    }


      public int compareTo(Object o) {

      if(o instanceof TestTransition)

      {

        TestTransition w = (TestTransition) o;

        if(this.getFromState().isReachable(this.getToState()))

        {

          if(w.getFromState().isReachable(w.getToState()))

          {

                return 0;

          }

          else

          {

                return 1;
```

```
            }
        }
        else if(w.getFromState().isReachable(w.getToState()))
        {
            return -1;
        }
    }
    return 0;
}
private int getNumberOfBitsforState()
{
        return FIX_NUMBER;
}

private int getNumberOfBitsforMethodNumber()
{
        return FIX_NUMBER;
}

private String getBoolStateString(State st)
{
                String retBoolVal = st.toBooleanString();
                if(retBoolVal.length()<getNumberOfBitsforState())
                {
                        retBoolVal =
NumberPorcessor.padeZeros(retBoolVal,getNumberOfBitsforState() -
retBoolVal.length(), true);
                }
                return retBoolVal;
    }
    private String getMethodBoolString()
    {
                String retBoolVal =Integer.toBinaryString(opt.getMethodNumber());
            retBoolVal =
NumberPorcessor.padeZeros(retBoolVal,getNumberOfBitsforMethodNumber()-
retBoolVal.length(), true);
            return retBoolVal;
    }


public State getFromState() {
        return preState;
}


public void setFromState(State preState) {
        this.preState = preState;
}


public State getToState() {
        return postState;
}


public void setToState(State postState) {
        this.postState = postState;
}
public String getMethodName()
{
        State st = this.getFromState();
        for(Transition t :st.getArNextStates())
        {
                if(t.getToState().isSameAs(this.getToState()))
                {
                        return t.getStrTitle();
                }
        }
        return null;
}
public ArrayList<InEqualitySimplified> getParameterConstraints()
{
        //ArrayList<InEqualitySimplified> retVal = new
ArrayList<InEqualitySimplified>();
        return
InequalitySolver.getOperationConstraintList(AFSMHolder.getOCLContext(), opt,
ConsType.PARAM);
}
public Object[] getParamValues()
{
        if(opt.getParameters().size()==0
||opt.getParameters().get(0).getName().trim().equals(""))
        {
                return new Object[0];
        }

        ArrayList<Object> retList = new ArrayList<Object>();
        for(InEqualitySimplified inq:getParameterConstraints())
        {
                System.out.println(inq.toString());
                retList.add(inq.getCurentValue());
        }

        return retList.toArray();

}
public Class[] getParameterTypes()
{
        System.out.println(opt.getParameters().size());
        if(opt.getParameters().size()==0
||opt.getParameters().get(0).getName().trim().equals(""))
        {
                return new Class[0];
        }
        System.out.println(opt.getParameters().get(0).getName());
        Class[] retParamTypes = new Class[this.opt.getParameters().size()];
        int index =0;
        for(CTMethodParameter param: opt.getParameters())
        {
                retParamTypes[index++] = param.getJavaType();
        }

        return retParamTypes;
}}
```

Appendix E

Java Code for Test Sequence Gene

```
package pk.com.rsoft.testsequenceoptimization.ga;


import java.io.Serializable;

import java.util.ArrayList;


import org.jgap.BaseGene;

import org.jgap.Configuration;

import org.jgap.Gene;

import org.jgap.InvalidConfigurationException;

import org.jgap.RandomGenerator;

import org.jgap.UnsupportedRepresentationException;


import pk.com.rsoft.classcontractstestbed.testsequences.TestTransition;

import pk.com.rsoft.classcontractstestbed.testsequences.TestSequence;

import pk.com.rsoft.classcontractstestbed.util.graph.AbstractFSM;


/**

 * @author Rehan Farooq

 */

public class TestSequenceGene extends BaseGene implements Gene,Serializable {

        private static final long serialVersionUID = 1L;

        private TestTransition theCall;


  public TestSequenceGene(TestTransition theCall, Configuration config) throws
InvalidConfigurationException

  {
        super(config);
        this.theCall = theCall;
    }
    @Override
    protected Object getInternalValue() {
        return theCall;
    }

    @Override
    protected Gene newGeneInternal() {
        try {
            return new TestSequenceGene(AFSMHolder.getRandomMethod(),
getConfiguration());
        } catch (InvalidConfigurationException ex) {
            throw new IllegalStateException(ex.getMessage());
        }
    }

    public void setAllele(Object a_newValue) {
        this.theCall = (TestTransition) a_newValue;
    }
    @Override
    public Object getAllele( )
    {
        return this.theCall;
    }

    public String getPersistentRepresentation() throws UnsupportedOperationException {
        try {
            return theCall.toBooleanString();
        } catch (InvalidConfigurationException ex) {
            throw new IllegalStateException(ex.getMessage());
        }
    }

    public void setValueFromPersistentRepresentation(String a_representation) throws
UnsupportedOperationException, UnsupportedRepresentationException {
        String [] parts = a_representation.split(",");
        if(parts.length<1 || parts.length>4)
        {
            throw new IllegalStateException("Invalid persistant representation :" +
a_representation);
        }
Double.parseDouble(yPart[1]), Double.parseDouble(zPart[1]),
Boolean.valueOf(parts[3]), s);
    }

    public void setToRandomValue(RandomGenerator a_numberGenerator) {
            theCall = AFSMHolder.getRandomMethod(a_numberGenerator);
a_numberGenerator.nextInt((int)s.getHeight()),
a_numberGenerator.nextInt((int)s.getLength()), theCall.getHECNStatus(), s);
    }

    public void applyMutation(int index, double a_percentage) {
        setToRandomValue(getConfiguration().getRandomGenerator());
    }
```

```
public int compareTo(Object o) {
  if(o instanceof TestSequenceGene)
  {
     return theCall.compareTo(((TestSequenceGene)o).theCall);
  }
  else
  {
    return -1;
  }
}
@Override
public String toString()
{
  try {
     return theCall.toBooleanString();
  } catch (InvalidConfigurationException ex) {
    throw new IllegalStateException(ex.getMessage());     }
}
     public static ArrayList<TestSequenceGene> toTestSequenceGenes(TestSequence
sequence,Configuration config) throws InvalidConfigurationException
     {
            ArrayList<TestSequenceGene> retList  = new
ArrayList<TestSequenceGene>();
            for(TestTransition call: sequence.getMethodCalls())
            {
                   retList.add(new TestSequenceGene(call, config));
            }
     return retList;}}
```

REFERENCES

References

1. Atul Gupta, "An Approach for Class Testing from Class Contracts", Springer 2010.
2. Thaise Yano et al, "Generating Feasible Test Paths from an Executable Model Using a Multi-Objective Approach", *ICSTW*, IEEE (2010).
3. Ruilian Zhao et al, "Empirical study on the efficiency of search based test generation for EFSM models" *3rd International Conference on Software Testing, Verification, and Validation Workshops*, IEEE (2010).
4. Mark Harman et al, "Optimizing for the Number of Tests Generated in Search Based Test Data Generation with an Application to the Oracle Cost Problem", *ISCTW*, IEEE (2010).
5. Andrea Arcuri et al, "Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing", IFIP International Federation for Information Processing (2010).
6. S. Asthana et al, "A Novel Approach to Generate Test Cases using Class and Sequence Diagrams", *IC3*, Springer (2010).
7. Shaukat Ali et al, "A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation", *IEEE Transactions on Software Engineering* (2010).
8. S.K. Prasad et al, "Optimization of Software Testing using Genetic Algorithm", *ICISTM*, Springer (2009).
9. S.K. Prasad et al, "An Ant Colony Optimization Approach to Test Sequence Generation for Statebased Software Testing", *ICISTM*, Springer (2009).

10. M. Prasannan and K.R. Chandran, "Automatic Test Case Generation for UML Object diagrams using Genetic Algorithm", *Int J. Advanced Soft Computing* ICSRS (2009).

11. T. Miller and P. Strooper, "A case study in model-based testing of specifications and implementations", Wiley Internet Science (2007).

12. Chen Mingsong et al, "Automatic Test Case Generation for UML Activity Diagrams", ACM (2006).

13. K. Derderian et al, "Automated Unique Input Output sequence generation for conformance testing of FSMs", *The Computer Journal*, ACM (2006).

14. Konak et al, "Multi-objective optimization using genetic algorithms: A tutorial", *Reliability Engineering and System Safety*, Elsevier, available at www.sciencedirect.com.

15. Marie-Claude Gaudel, "Testing from Formal Specifications, a Generic Approach", Springer (2001).

16. Ying Gao, "Study on multi-objective genetic algorithm", IEEE (2000).

17. James A. Whittaker, "What Is Software Testing? And Why Is It So Hard?", IEEE Software (February 2000).

18. S. R. Dalal et al, "Model-Based Testing in Practice", Proceedings of ACM ICSE (1999).

19. Murata, T., "MOGA: multi-objective genetic algorithms", IEEE (1995).

20. F. P. Brooks, "No Silver Bullet Essence and Accidents of Software Engineering", IEEE (1987)

21. Object Management Group (OMG)'s Formally Released versions of Object Constraint Language (OCL) available at http://www.omg.org/spec/OCL/

22. The Dresden OCL parser version 3.1. available at http://www.dresden-ocl.org/

23. Java Genetic Algorithm Package (JGAP) available at http://jgap.sourceforge.net

24. Metaheuristic Algorithms in Java (jMetal) an API for GA and MOGA based optimization, available at http://jmetal.sourceforge.net/

25. Eclipse IDE for Java Developers, available at http://www.eclipse.org/downloads/

26. JUnit a Unit Testing Framework for Java, available at http://www.junit.org/

27. Java Reflection API for assessing runt time properties of Java Objects java doc available online at http://docs.oracle.com/javase/tutorial/reflect/index.html

28. Mutation Java (μJava), a mutation system for Java programs, version 3 download and documentation available at http://cs.gmu.edu/~offutt/mujava/

International Islamic Univ.
CENTRAL
LIBRARY
ISLAMABAD.