Edith Cowan University
Research Online

ECU Publications Pre. 2011

2009

Evolving the Quality of a Model-Based Test Suite

Usman Farooq
Edith Cowan University

Chiou P. Lam
Edith Cowan University

This article was originally published as: Farooq, U. , & Lam, C. P. (2009). Evolving the Quality of a Model-Based Test Suite. Proceedings of International Conference on Software Testing, Verification and Validation. (pp. 141-149). USA, Denver. IEEE Computer. Original article available here
This Conference Proceeding is posted at Research Online.
http://ro.ecu.edu.au/ecuworks/172

IEEE International Conference on Software Testing Verification and Validation Workshops

Evolving the Quality of a Model Based Test Suite

Usman Farooq, C. P. Lam
School of Computer and Information Science
Edith Cowan University
Perth, Australia
ufarooq@student.ecu.edu.au, c.lam@ecu.edu.au

***Abstract*— Redundant test cases in newly generated test suites often remain undetected until execution and waste scarce project resources. In model-based testing, the testing process starts early on in the developmental phases and enables early fault detection. The redundancy in the test suites generated from models can be detected earlier as well and removed prior to its execution. The article presents a novel model-based test suite optimization technique involving UML Activity Diagrams by formulating the test suite optimization problem as an Equality Knapsack Problem. The aim here is the development of a test suite optimization framework that could optimize the model-based test suites by removing the redundant test cases. An evolution-based algorithm is incorporated into the framework and is compared with the performances of two other algorithms. An empirical study is conducted with four synthetic and industrial scale Activity Diagram models and results are presented.**

Keywords-Model Based Testing; Test Suite Optimization; UML;

I. INTRODUCTION

Software development is generally a manual and error-prone process. Anomalies and bugs can occur at any stage and likely to have serious consequences on quality, cost and schedule of the development of the software. Ideally, these should be detected and corrected early on in the development phases. However, in reality, software requirements often evolve through modification and refinement processes owing to their ambiguities and incompleteness. In such cases, updating test suites to meet evolving requirements is quite complex. Removing obsolete test cases, modifying obsolete test cases or generating new test cases as a consequence of the changes to requirements can be laborious and chaotic. Model-Based Testing (MBT) is more appropriate than conventional testing because of its high potential for automation, ease of accommodating changes and maintenance [1, 2]. Models are intuitive for visualizing and analyzing complex systems. While they are developed initially for capturing the information about the software system, they also have the advantage of being reused as the development progress. New or modified requirements only necessitate an updating of the models. The process of model based test case generation is basically a traversal of the models. Automatic on-the-fly generation of test cases simplifies the maintenance of a test suite. Conventional testing is usually performed towards the end of software development and is known as a single phase. MBT allows testing activities during the earlier development phases and leverages control points i.e. requirement and design validation at various levels.

Test case generation is the most demanding and crucial of testing activities [3, 4]. Usually test suites are generated according to some given objective criteria. The same test case generation criteria i.e. coverage and fault-based criteria, are also used to evaluate the quality of the test suite. In coverage-based techniques, the quality of the test suite is determined by the percentage of the code execution resulting from using the test suite.. With a fault-based technique the test suite is considered adequate if it detects all of the injected faults in the software. The test generation mechanism adopted to meet the coverage or fault criteria can be deterministic or stochastic. Deterministic techniques are effective but are complex and computationally expensive to apply. Stochastic test generation techniques are automatic, simple and easy to use but are less efficient as they produce many redundant test cases.

The model-based random testing technique is agile, immune to the pesticide-paradox [5], and is characterized by its simplicity and readiness efficacy. Owing to the stochastic nature of this technique, the probability of newly generated test cases not addressing some undetected defects is less likely. MBT's readiness stems from the reusable models and inexpensive simple techniques that test cases can be generated promptly whenever they are needed. While this technique can produce as many test cases as one needs, ironically it can pollute the test suite with an inordinate number of unintentional redundant test cases. A test suite with redundant test cases increases the test suite size, takes far longer to complete without providing any obvious advantage or enhanced confidence. The additional time and effort needed to execute these unwarranted test cases or to analyze the failure of redundant test cases obviously raises the testing cost, diminish overall productivity and waste often-scarce project resources. Moreover, redundant test cases reduce the quality of a good test suite as Kaner, Falk and Nguyen (1993) have stressed that a good test case does not waste the scarce time and resource in serving the same testing purpose as another test case [6].

978-0-7695-3671-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICSTW.2009.27

IEEE computer society

The process of identification and removal of redundant test cases that finally yields a minimal test suite can be defined as test suite optimization. As redundancy of a test case is relative and is dependent on the test criteria and the other test cases in a test suite, the optimization process may need to evaluate all possible combinations of the test cases in a test suite and calculate their cumulative coverage. For a test suite with *n* test cases, the number of evaluations will be the order of *n* potential combinations. The process of manual identification and removal of redundant test cases is both overwhelmingly complex and erratic. Similarly, exhaustive analysis even with an automated tool would handle only relatively trivial test suites and is deemed impractical for industrial-scale test suites. The complexity of test suite optimization problem is exponentially related to the original test suite size. Thus, because of this combinatorial explosion problem, test suite optimization cannot be attained in polynomial time except for a trivial test suite.

Evolutionary Computation (EC) is a class of metaheuristic techniques that is based on the natural process of evolution and has proven to be an effective search process. It has been successfully applied to various research and application fields such as combinatorial optimization, neural nets evolution, planning and scheduling, industrial design, management and economics, machine learning and pattern recognition. For application, initially a problem is defined as an optimization problem and then a set of potential solutions are encoded using some coding scheme. New solutions are generated using nature inspired reproduction function. The technique merely needs the fitness function to evaluate the individual solutions and to guide the underlying heuristic.

The Unified Modeling Language (UML) is a de-facto industry standard for object oriented analysis and design of software systems. UML2 (revision 2 of UML) now have 13 diagrams (6 Structural and 7 Behavioural) and each diagram is a collection of tightly coupled modelling concepts with an ability to represent a specific aspect of the system. A UML model is developed to depict a chosen viewpoint of the systems according to the underlying language formalism. For example, the State Machines language elements enable modellers to specify discrete event-driven behaviour using a variant of the well-known statecharts formalism, and the Activities language elements provide for modelling behaviour based on a flow-oriented paradigm. Activity Diagram (AD) is a behavioral type of diagram supporting control and data flow modeling of the system. In UML2, AD introduces several concepts i.e. branching, concurrency, synchronization and token flow semantic that make it ideal for modeling complex systems.

In this paper, we deal with the problem of finding a subset of minimal size without redundant test cases by reformulating this problem as an Equality Knapsack Problem. We demonstrate the test suite optimization through an example. An empirical study was conducted with industrial-scale models and results were compared with those produced by other algorithms. We define a test case as being redundant, in accordance with a specific criterion, that is, if it fails to add extra information or coverage. We hypothesized that the elimination of redundant test cases, according to specific coverage criteria, could optimize the test suites and potentially save scarce project time and resources. Until now, almost every published study involving evolutionary testing has focused on code-based test case generation and prioritization. To our knowledge, this is the first attempt to formalize and automate UML based test suite optimization with an evolutionary metaheuristic.

The paper is organized as follows. Section 2 describes the test suite optimization problem and the evolution-based test suite optimization technique is introduced in section 3. Experiment, corresponding results and discussion are presented in Section 4. Related work, summary and future work are provided in Sections 5 and 6.

II. Test Suite Optimization

A. Formal Definition

A model-based test suite is given in the form of set TS with elements a_i, size n and coverage m. The set elements a_i are test cases where each test case is a sequence of model elements representing an execution path in the model. The coverage m is calculated by the percentage of model elements required by test criteria that have been executed by the given test suite. The size n is the number of test cases in the test suite. The objective is to find a minimal subset $ts = \{a_1, a_2, \dots, a_n\}$ in such a way that $m_{ts} = m$.

B. Illustrated Example

A test suite generated for branch coverage involves a criterion requiring at least one test case for each branch that will cause its execution. Unfortunately, a test suite generated using a stochastic technique with this property contains many redundant test cases. To illustrate the problem further, a test suite is generated as shown in table-1 using a stochastic test sequence generation (TSG) algorithm proposed in [7] for an example model as shown in figure-1. The generated sequences of model constructs, formally referred here as paths, are usually evaluated according to a specified criterion.

The generated test suite is analyzed w.r.t. a UML 2.0 AD based branch coverage criteria as defined in [7]. The columns, for example, ‘e2’ and ‘e3’, indicate the branches in the model. Likewise, the column names are abbreviated ‘BC’, ‘Cov.’ and ‘RCov’ for branch coverage (number of branches covered), coverage (percentage) and running coverage respectively. The notion of 0 and 1 indicates missing and executing branches in a test case, respectively. The test suite has 20 test cases and 100 percent coverage w.r.t. the branch coverage. The suite in its current form is not optimized as it has many redundant test cases.

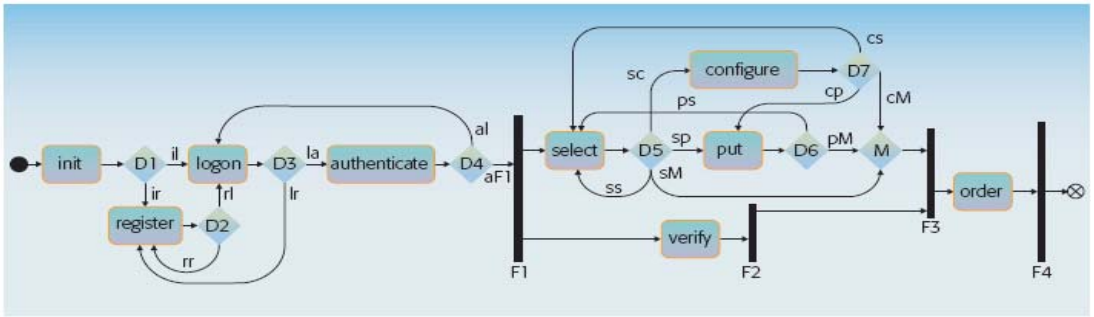

Figure 1: AD model of an Enterprise Customer Commerce System (ECCS) [8]

Table 1: Optimized Test Suite for an ECCS

TC	Branches																	BC	RBC	RCov
	e2	e3	e5	e6	e8	e9	e11	e12	e16	e17	e19	e24	e21	e22	e23	e26	e27			
1	1	0	0	0	0	1	0	1	0	0	0	1	0	0	0	0	1	5	29.41	29.41
2	1	0	0	1	1	1	0	1	0	0	0	1	0	0	0	0	1	7	41.18	41.18
3	1	0	0	0	0	1	0	1	1	0	0	1	0	0	0	0	1	6	35.29	47.06
4	1	0	1	1	1	1	1	1	0	0	1	0	1	0	0	0	0	9	52.94	70.59
5	1	0	0	0	0	1	0	1	1	0	1	0	1	0	0	0	0	6	35.29	70.59
6	1	0	1	1	1	1	0	1	0	0	0	1	0	0	0	0	1	8	47.06	70.59
7	1	0	1	1	1	1	1	1	1	0	1	1	0	1	0	0	1	12	70.59	76.47
8	1	0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	5	29.41	82.35
9	1	0	1	1	1	1	0	1	1	0	0	1	0	0	0	0	1	9	52.94	82.35
10	0	1	0	1	0	1	1	1	0	0	1	0	1	0	0	0	0	7	41.18	88.24
11	1	0	1	1	1	1	1	1	1	0	1	0	1	0	0	0	0	10	58.82	88.24
12	0	1	1	1	1	1	1	1	0	0	1	0	1	1	0	0	0	10	58.82	88.24
13	0	1	1	1	0	1	0	1	0	1	0	1	0	1	1	0	0	9	52.94	94.12
14	1	0	0	0	0	1	0	1	0	0	1	0	1	0	0	0	0	5	29.41	94.12
15	1	0	1	1	1	1	1	1	0	1	0	1	0	0	0	1	0	10	58.82	100.0
16	1	0	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	9	52.94	100.0
17	1	0	1	1	1	1	0	1	1	1	0	1	0	1	1	1	0	12	70.59	100.0
18	0	1	1	1	0	1	0	1	0	1	1	0	0	1	0	0	0	8	47.06	100.0
19	0	1	1	1	1	1	0	1	0	0	1	0	1	0	0	0	0	8	47.06	100.0
20	0	1	1	1	1	1	1	1	0	0	1	1	1	1	1	0	0	12	70.59	100.0

Table 2: Optimized Test Suite for an ECCS

TC	Branches																	BC	RBC	RCov
	e2	e3	e5	e6	e8	e9	e11	e12	e16	e17	e19	e24	e21	e22	e23	e26	e27			
7	1	0	1	1	1	1	1	1	1	0	1	1	0	1	0	0	1	12	12	70.59
10	0	1	0	1	0	1	1	1	0	0	1	0	1	0	0	0	0	7	14	82.35
17	1	0	1	1	1	1	0	1	1	1	0	1	0	1	1	1	0	12	17	100.00

According to the definition specified earlier, test cases 5, 6, 9, 11, 12, 14 and 16–20 appear to be redundant. By removing them, the test suite can achieve the same coverage with only 9 test cases (all non-redundant test cases in the original test suite). However, from table-1, it can be seen that some of the test cases have a higher coverage than others but many are redundant as they failed to improve the overall coverage. For instance, the test case TC-7 has a higher coverage than test cases TC1– TC3 but it also subsumed the nodes that these three covered. Similarly, test case TC-17 has doubled the coverage of TC-8 and also subsumed its covered nodes. Without changing the execution order of the test cases and by skipping the redundant test cases, the consolidated coverage of both TC-7 & TC-17 is more than 85% and reduces the size of the test suite by 80%. More interestingly, the combination of just three test cases TC-7, 10 & 17 provides complete coverage with no redundant test cases in the test suite as shown in table 2. In the optimized test suite, the achieved coverage is constant regardless of the execution order of the test cases.

C. Formulation as a Equality Knapsack Problem

The knapsack problem is a class of combinatorial optimization problems that has been extensively studied. In the basic version, the knapsack has some specific capacity and a set of objects with a given weight and profit. The problem is defined as finding a set of objects, such that the total profit of the set is maximized without exceeding the knapsack capacity. There are many knapsack variants including the Equality Knapsack problem (EKP) where the objective is to find a subset from a given set of items in such a way that the total profit is maximized and the total weight c is exactly equal to the given capacity C [9].

The test suite optimization problem can be translated into EKP. For instance, a test suite has n test cases that correspond to objects in the knapsack problem and the coverage C of the test suite corresponds to knapsack capacity. Each test case i has coverage c_i that corresponds to the weight of an object. In order to show the inclusion or exclusion of a particular test case, a binary decision variable x is used. The requirement to be satisfied is

$$\sum_{i=1}^{n} c_i x_i = C \qquad (1)$$

$$where\ x_i \in \{0,1\}, i = 1, \dots, n. \qquad (2)$$

The utility value of a test case in the test suite that corresponds to cost in the knapsack problem is $u = 1$for each test case. The objective is to find a test suite at a given coverage in such a way that the total cost of the test suite is minimized. Since the capacity of the test suite is C, we require that the total weight of all chosen test cases (the total weight of the generated test suite) is

$$f(x) = \sum c_i x_i$$

to be C exactly. As the EKP can be formulated into a minimization version by minimizing the cost of the items in the knapsack, so formally the problem can then be stated as

$$minimize \sum_{i=1}^{n} u_i x_i \qquad (3)$$

III. Evolution based Test Suite Optimization

Evolutionary Computation is a metaheuristic, inspired by the natural process of evolution. Genetic Algorithm (GA) and Evolutionary System (ES) are two basic EC algorithms and differ in their emphasis on optimization procedures and problem representation. In the basic GA, the crossover is a primary reproduction operator to breed offspring, mutation is used to alter one or more allele values and designed to solve discrete or integer optimization problems. ES was originally applied to solve continuous parameter optimization problems and mutation was used as a main operator for reproduction. Nevertheless in this paper, we retain both GA and ES while evolving the test suite optimization as both schemes are often hybridized in real world applications, according to the requirements of the application domain.

A. Framework Design

The incorporation of EC in test suite optimization involves several careful design decisions i.e. problem encoding, selection and design of operators for solution production, formulation of a fitness function to evaluate the quality of these solutions to guide the underlying heuristic search and then refining the operating parameters to enhance the performance of the algorithm.

The first and foremost step with the application of evolutionary algorithm is the representation of the problem domain into a particular coding scheme i.e. binary, real value and etc. The encoding scheme defines the search space and links the genotype to a corresponding phenotype. The effect of encoding is very crucial as the entire search operations are performed only on the representation that abstracts the individual parameters. Similar to the knapsack-problem, binary encoding is considered to be a direct and natural representation for the test suite optimization. Test suites are directly encoded in the form of genotype. The inclusion and exclusion of a test sequence within a test suite are represented by 1 and 0 respectively in a binary sequence (chromosome string). So, a randomly generated sequence of 0 and 1 represents a test suite with a particular combination of test sequences. The total number of 1's in a binary sequence (chromosome) represents the size of the test suite. In order to determine the phenotypic properties i.e. size and coverage for an individual (test suite), the number and collective coverage of all the included test cases are calculated respectively. The total number of 1's in a binary

sequence (chromosome) represents the size of the test suite or the total number of test cases in a test suite.

The notions of a better or inferior solution and a fitness measuring mechanism have pivotal roles in evolutionary optimization as they guide the underlying search mechanism. As the objective of test suite optimization is to search for a minimal combination of test cases from the original test suite, equation (3) is used to evaluate the fitness of each candidate solution. A precondition of equation (3) requires that a valid candidate solution must satisfy equation (1).

The design of selection and production functions is also crucial to the adaptation of EC. The selection function defines rules for the selection of sub-population (mating pool) for the production of future offspring. Various selection rules i.e. rank, probabilistic, fitness-proportionate and tournament selection are widely used in EC applications. We opted to use tournament selection owing to its robustness and convenience [10]. The replacement mechanism defines the placement of offspring into the population and for that we used a steady-state technique. The production function to breed new individuals comprises both recombination and mutation operators. Typically, the recombination operation can be either sexual or asexual. The sexual reproduction a.k.a. crossover produces new offspring from the parents. The individuals selected according to their fitness for mating survive through the generations and propagate their characteristics into the offspring. Therefore, through crossover the search converges towards the promising regions of the search space. The mutation operation introduces noise and prevents premature convergence of the search process to local optima by randomly sampling new points in the search space. In terms of bit strings, mutation is applied by inverting bits at random within a string with a certain probability called the mutation rate which defines the number of bits that will be flipped at each iteration step. Similarly, the crossover mechanism essentially breeds new solutions by swapping the substrings of existing solutions (test suites) at each iteration step. In this paper, a double-point crossover operator and a single-point mutation operator are used.

IV. Empirical Study

The experiment is designed with an objective to verify a number of research questions which are listed below:

Research Question 1: Reformulating the problem of removing redundant test cases as a combinatorial optimization problem can reduce test suite size; which generalized optimization technique (i.e. EC, Greedy and Hill Climbing) is more effective for model-based test suite reduction.

Research Question 2: How does the test suite size affect the optimization of the test suite?

Research Question 3: How does the order of test cases affect the optimization of the test suite?

A. Experimental Setup

As the optimization techniques attempt to reduce the test suite cost w.r.t. a given coverage criterion so the percentage reduction will be used as a surrogate measure for comparative analysis. For testing our hypothesis, we conducted the experiment with four models of varying sizes and complexity levels. The AD model shown in figure 1, describes an Enterprise Customer Commerce System (ECCS) taken from [8]. It describes the process of online purchase of products that is comprised of two sub-processes: authentication and shopping. The first process authorizes existing users for shopping and account configuration. However, in the case of new customers, it enables them to register first. The shopping process facilitates the user to order selected products and to configure his/her account if required. The Automatic Teller Machine (ATM) model is a popular case study. For our experiment, we adapted it from a report [11]. The ATM model comprises of an activity diagram with a top level view of ATM operations which are further elaborated as low level AD diagrams with details of the operations i.e. withdraw cash, deposit money, transfer funds and check balance. The experiment also includes two industrial scale AD models of a module in an Intelligent Transport System (ITS), namely Edit Trend Properties (ETP) and Delete Trend Properties (DTP). Both models respectively describe the step by step editing and deletion of existing trending reports from archived or real-time data. For more details of each model see table 3. The columns i.e. Nodes, Branches and complexity indicate the size and cyclomatic complexity of the studied models respectively.

Table 3: Characteristics of sample models

Model	Nodes	Branches	Edges	Complexity
ECCS	23	17	33	11
ATM	135	28	141	16
ETP	77	26	89	14
DTP	52	37	57	21

Table 4: Parametric settings of EC for the experiment

Parameters	**Value**
Objective	Minimize test suite size
Population size	50
No. of Generations	50
Replacement scheme	Steady state
Crossover rate	0.9 (double point)
Mutation rate	0.2 (single point)
Selection Scheme	Pair wise tournament

In order to obtain redundant test cases, a stochastic test sequence generation technique is used [7]. Three sets of test suites relatively larger in size are generated using this random walk-based algorithm for each model. These are then evaluated according to the model-based branch coverage criterion. Using the optimization framework proposed in section 4, the generated test suite is optimized. Considering the stochastic nature of the evolutionary metaheuristic, each experiment is repeated 10 times. The initial population is randomly generated and the associated parametric values used initially are as suggested in [10]. However, the parametric values are subsequently refined to improve the performance of the algorithm. The final parametric values for EC algorithm are shown in table 4. The results of the proposed framework are compared with those of Greedy and Hill Climbing algorithms. A Greedy algorithm is easy to design as it builds the solution step by step according to a given objective function. The algorithm always chooses current best solution without considering the future affect. The pseudo code for Greedy algorithm is presented in fig. 4. Hill Climbing is a local search technique that begins from a randomly selected initial solution in the search space and then iteratively improves the solution until the termination condition is met. It is one of the simplest optimization algorithms. The pseudo code for the Hill Climbing is given in fig. 2.

```
Randomly select currentItem in the search space;
Until nIteration ≤ maxIterations do
  neighbours = getNeighbours(currentItem);
  nextItem = getBestNeighbour(neighbours);
  if Evaluate(nextItem) ≥ Evaluate(currentItem) then
      currentItem = nextItem;
end
```

Figure 2: Hill Climbing Algorithm Pseudo Code

```
Initialize population randomly;
  Until nGeneration ≤ maxGenerations do
    for each individual in population do
      Evaluate the fitness;
    end
    Select two best individual to mate, p1 & p2;
    Offspring = Crossover (p1, p2);
    Mutate (offspring);
    Replace the offspring in the population;
    if Stagnation Condition is satisfied then
      return;
  end
```

Figure 3: Evolutionary Computation Algorithm Pseudo Code

```
currentItem = 0;
  sort(item-List);
  for each item in Item-List do
     if (Evaluate(currentItem) ≥ Evaluate(item)) then
        currentItem = item;
  end
```

Figure 4: Greedy Algorithm Pseudo Code

Although for performance analysis of algorithms various measures (i.e. time and space) can be used. However, in this study we compare the algorithms in terms of their efficiency with test suite reduction and for that we performed following statistical tests: the One-Way ANOVA, Correlation Analysis and Paired T-test.

B. Results and Discussion

The results of the experiments are presented in Table 5. The column names HC, GD and EC represent Hill Climbing, Greedy and Evolutionary Computation framework respectively. These results illustrate three significant observations: (1) Significant reduction in the size of most of the test suites without affecting their effectiveness, (2) Consistent and scalable evolutionary test suite optimization and (3) better performance produced from using EC than those from Greedy and Hill Climbing algorithms in most cases. From observations of the data, the reduction in the size of the test suites is quite obvious but to gain confidence we applied Paired Samples T-test to confirm any significant differences. In all cases, the final optimal test suite has the same coverage level as in the original test suite.

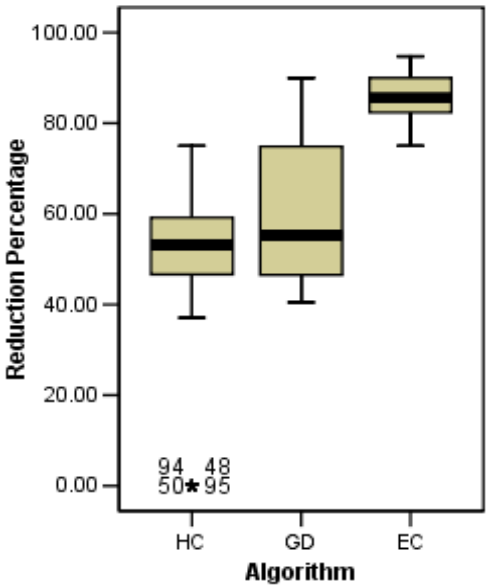

Figure 5: Box-plot for test suite reduction by different algorithms

Table 5: Size Reduction w.r.t. Branch Criterion

Model	Un-Optimized TS Size	Optimized TS (Size reduction %)								
		Average			Minimum			Maximum		
		HC	GD	EC	HC	GD	EC	HC	GD	EC
ECCS	20	65	85	85	65	85	85	75	85	85
	25	64	88	88	52	88	88	68	88	88
	30	66.7	90	90	53.3	90	90	70	90	90
ATM	89	37.1	48.3	91	0	48.3	89.9	47.2	48.3	92.1
	111	47.8	45.1	92.8	0	45.1	91.9	55.9	45.1	93.7
	133	46.6	46.6	94	0	46.6	93.2	54.9	46.6	94.7
ETP	27	0	59.3	77.8	0	59.3	77.8	51.9	59.3	77.8
	34	55.9	64.7	82.4	52.9	61.8	82.4	58.8	64.7	82.4
	41	53.7	61	85.4	41.5	61	85.4	56.1	61	85.4
DTP	28	50	46.4	78.6	42.9	42.9	75	57.1	46.4	78.6
	35	51.4	48.6	82.9	48.6	42.9	80	57.1	51.4	82.9
	42	57.1	42.9	85.7	50	40.5	85.7	64.2	50	85.7

The test suite reduction by EC for each model is more than 75% which is quite remarkable. The data substantiates the stability and robustness of the proposed evolutionary framework for model-based test suite optimization in comparison to the Hill Climbing and Greedy algorithms. The results of Post-Hoc Tukey HSD (pair-wise) comparison of EC-Greedy and EC-Hill Climbing show significant differences at 99% confidence interval. It can be inferred that the EC on average performs better than both Greedy and Hill Climbing algorithms. Figure 5 provides an insight into the performance of each algorithm w.r.t. reduction percentage of size of test suite.

The average reduction in percentage of size from using the Hill Climbing, Greedy and EC algorithms are approximately 53, 55 and 86 respectively. Although, the average performance of the Greedy algorithm appears to be only slightly better than that of the Hill Climbing, the difference between the two is still statistically significant**.** The larger spread of data for the Greedy algorithm and outliers for the Hill Climbing algorithm confirms the known issues with these two algorithms i.e. inconsistent and un-scalable performance. We infer that the inconsistency in the performance of the Greedy algorithm is due to its iterative, non-exploratory, solution construction mechanism which often makes it converge to a non-optimal solution. Although the Greedy algorithm is quite fast in generating solutions, in most cases it fails to find the global optimum. Hill Climbing often gets trapped into a local optimum, and even in some cases it failed to improve the initial solution (see outliers 48, 50, 94 and 95 in figure 5) due to the flat section in the search space.

In order to see the effect of the ordering of test cases on test suite optimization, each test suite was randomly shuffled 5 times. The median test analysis reveals interesting aspects about optimization involving each algorithm. The change in the ordering of test cases does not have a significant difference on the test suite optimization. Moreover, despite a change in search landscape, the performance of EC and GD was not changed significantly. However, the reduction in test suite with HC seems to be greatly affected by the ordering of the test cases. Although the performance of EC was quite consistent, however in some cases it could not find the global optimal (see column 'minimum' in table 5). There could be many reasons i.e. population size, premature convergence and maximum generations, however due to the limitations of the experiments that we have conducted so far, we cannot infer any further. Subsequent experimentations will be required to explore this aspect.

The increase in test suite size was expected to increase the redundancy; however the reduction in the size of test suite is not reflected in same proportion in the HC and GD algorithms. In order to see if there is any interaction of test suite size onto the optimization process, we applied correlation analysis. The analysis shows the following statistically significant results: strong positive association between EC and test suite size, strong negative relationship between test suite size and GD and moderate negative correlation between test suite size and HC.

C. Threats to Validity

Although a great deal of care was taken in the study to avoid any bias, the following are some of the potential issues that can undermine the results and subsequently the conclusions.

Experiments were conducted with models which were selected from various sources and used without any modifications. Despite these models representing a diverse set of size and complexity; they are not overly large or complex. So, one of the threats to the generalization of the

results presented here is the limited classes of models. Using larger and more diverse/complex models will expand the sample space and may affect the results.

We performed the study using only AD models which are ideal candidates for path-based testing due to their flow-based semantics. We did not include other UML behavioral diagrams i.e. state machine and sequence diagram in the study, nor did we use any structural diagrams. Thus the results presented in this paper cannot be generalized to the other UML diagrams.

Another threat to the generalization is related to the peculiarity of the stochastic test generation technique and its implementation used here. One of the shortcomings of the said technique relates to the redundancies in the generated test suite. The results described in the paper are specific to the stochastic test generation technique used here and other stochastic test generation techniques or implementation may produce different results.

Finally, the only factor used to optimize a test suite is branch coverage. Other types of potential optimization factors including coverage criterion, mutation score and cost can yield different results. Even various combinations of these potential factors may produce different results.

V. Related Work

Redundancy in test suites is generally not desirable as it wastes project resources and increases the cost of testing. We position that the elimination of redundant test cases according to a specific criterion would optimize the test suite. The work related to our study can be classified into two categories: (1) optimization of test suites, and (2) application of metaheuristic techniques in test suite minimization/prioritization.

Test suite with a large number of redundant test cases is often considered inefficient and various researchers have tried to tackle this problem. Chen and Lau proposed a divide-and-conquer approach to minimize the size of a test suite generated through a random technique [12]. It is based on an exact algorithm, which is generally considered infeasible for real world application. Xie *et al.* have also developed a framework for the optimization of object oriented unit tests [13] by eliminating redundant test cases. Authors also proposed a number of redundancy detection approaches and applied it in detecting and removing redundant test cases. Jeffrey and Gupta proposed a technique to minimize test suite with selective redundant test cases [14]. Harrold, Gupta and Soffa proposed a code based heuristic technique to remove obsolete and redundant test cases from an original test suite and to obtain a reduced test suite [15].

Tallam and Gupta adapted greedy algorithm to minimize a test suite by removing redundant test cases [16]. They employed the Concept Analysis technique to identify groups of objects and their attributes and implications and then exploit this information for test suite reduction. Heimdahl and George has investigated the effects of test suite reduction for formal specification based test suites [17].

The analogy between the test suite optimization and combinatorial optimization was defined and investigated by Harman and Jones [18]. A large portion of research reported with the application of metaheuristic techniques in software testing is focused on test case prioritization and code-based techniques [19]. Shin and Harman (2007) formulated the test case selection as a multi-objective problem with a provision to select a test case subset according to the given two, three or more objectives [20]. Their study found that although the evolutionary techniques produce larger pareto front than the additional greedy algorithm however in terms of performance they are not significantly different. Li, Harman and Heirons evaluated various heuristic algorithms for regression test case prioritization [21]. Their study concludes that the genetic algorithm is equivalent to greedy algorithms in terms of performance and even more suitable for situations where the fitness of the test suite is not predetermined. The work presented in this paper is different from their work in two ways. First, our approach aims to minimize the generated test suite and secondly, it is focused on UML model based testing as compared to code based testing used in their work.

Until now most of the research reported with the application of metaheuristic techniques was focused on test case prioritization and code-based techniques. The optimization of test suite is a controversial topic, mainly because of the varying reports on the fault detection capabilities of the reduced test suite [22-24]. So far, the proposition about the fault detect-ability of optimized test suite is limited to the code-based regression test suite (For more detail please see [24, 25]). One study in the category of model based testing regarding test suite minimization and fault detection capability was conducted with formal specification [17]. However, due to the enormous differences between modeling techniques and the associated test generation mechanisms those results are not necessarily applicable to other model based techniques. However, this fact highlights the need for further study.

VI. Summary

The field of search-based software engineering is new and the incorporation of various metaheuristic techniques has heralded a new era of research and development. In software testing and particularly in structural testing many researchers have successfully incorporated these techniques for test data generation and to regression test suite prioritization. However, in model based testing and specifically UML based testing there is still much need to be done. The work reported in this paper has two contributions: (1) the formulization of test suite minimization as a combinatorial optimization problem and (2) the

development of an optimization framework for UML model based test suites. We demonstrated the feasibility of the technique with an empirical study. The experimental results show the robustness of the proposed technique that optimizes the test suites generated from AD model w.r.t. the branch coverage criterion.

REFERENCES

[1] A. Pretschner, "Model-Based Testing in Practice," in *Formal Methods (FM'05)*. vol. 3582: Springer-Verlag, 2005, pp. 537-541.

[2] I. K. El-Far and J. A. Whittaker, "Model-Based Software Testing," in *Encyclopedia of Software Engineering*, 2nd ed, J. J. Marciniak, Ed.: John Wiley & Sons, Inc., 2002.

[3] A. Bertolino, "Software Testing Research and Practice," in *10th International Workshop on Abstract State Machines (ASM'2003)*, Taormina, Italy, 2003, pp. 1-21.

[4] M. Ould and B. Praxis, "Testing-a challenge to method and tool developers.," *Software Engineering Journal,* vol. 6, pp. 59-64, 1991.

[5] B. Beizer, *Software testing techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.

[6] C. Kaner, J. Falk, and H. Q. Nguyen, *Testing Computer Software* 2nd ed.: International Thomson Computer Press, 1993.

[7] U. Farooq, C. P. Lam, and H. Li, "Towards Automated Test Sequence Generation," in *19th Australian Software Engineering Conference (ASWEC'2008)*, Perth, Australia, 2008

[8] J. M. Küster, J. Koehler, and K. Ryndina, "Improving Business Process Models with Reference Models in Business-Driven Development," in *Business Process Management Workshops*, 2006.

[9] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems*: Springer, 2004.

[10] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*: Addison-Wesley, 1989.

[11] R. Chandler, C. P. Lam, and H. Li, "UML Models with Activity Diagrams: for Case Studies," SCIS, Edith Cowan University, Perth, Technical Report TR-SERG-06-02, 2006.

[12] T. Y. Chen and M. F. Lau, "On the divide-and-conquer approach towards test suite reduction," *Information sciences,* vol. 152, pp. 89-119, June 2003 2003.

[13] T. Xie, D. Marinov, and D. Notkin, "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests," in *Automated Software Engineering*, Linz, Austria, 2004, pp. 196-205.

[14] D. Jeffrey and N. Gupta, "Test Suite Reduction with Selective Redundancy," in *International Conference on Software Maintenance (ICSM 2005)*, Budapest, Hungary, 2005, pp. 549-558.

[15] M. J. Harrold, R. Gupta, and M. L. Soffa, "A methodology for controlling the size of a test suite," *ACM Trans. Softw. Eng. Methodol.,* vol. 2, pp. 270--285, 1993.

[16] S. Tallam and N. Gupta, "A Concept Analysis Inspired Greedy Algorithm for Test Suite Minimization," in *Workshop on Program Analysis for Software Tools and Engineering*, Lisbon, Portugal, 2005.

[17] M. P. E. Heimdahl and D. George, "Test-suite reduction for model-based tests: Effects on test quality and implications for testing," in *Automated Software Engineering* Linz, Austria, 2004, pp. 176-185.

[18] M. Harman and B. F. Jones, "Search based software engineering.," *Information and Software Technology,* vol. 43, pp. 833-839, December 2001.

[19] M. Harman, "The Current State and Future of Search Based Software Engineering," in *Int. Conference on Software Engineering, Future of Software Engineering (FOSE'07)* Minneapolis, USA: IEEE, 2007.

[20] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *International Symposium on Software Testing and Analysis*, London, United Kingdom, 2007, pp. 140-150.

[21] Z. Li, M. Harman, and R. Heirons, "Search Algorithms for Regression Test Case Prioritisation " *IEEE Transactions on Software Engineering. ,* vol. 33, pp. 225-237, 2007.

[22] T. Y. Chen and M. F. Lau, "Test Suite Reduction and Fault Detecting Effectiveness: An Empirical Evaluation," in *Reliable Software Technologies: Ada Europe 2001* Leuven, Belgium, 2001, pp. 253-265.

[23] W. E. Wong, J. R. Horgan, L. London, and A. P. Mathur, "Effect of Test Set Minimization on Fault Detection Effectiveness," in *17th International Conference on Software Engineering*, 1995, pp. 41-50.

[24] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault-Detection Capabilities of Test Suites," in *International Conference on Software Maintenance (ICSM 1998)* Bethesda, MD, 1998.

[25] W. E. Wong, J. R. Horgan, A. P. Mathur, and A. Pasquini, "Test set size minimization and fault detection effectiveness: A case study in a space application " *Journal of Systems and Software,* vol. 48, pp. 79-89, 1999.