Summer 2013

Hybridizing and applying computational intelligence techniques

Jeffery Scott Shelburg

Department:

HYBRIDIZING AND APPLYING COMPUTATIONAL INTELLIGENCE TECHNIQUES

by

JEFFERY SCOTT SHELBURG

A THESIS

Presented to the Faculty of the Graduate School of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE IN COMPUTER SCIENCE

2013

Approved by

Dr. Daniel Tauritz, Advisor
Dr. Marouane Kessentini
Dr. Samuel Mulder

Sandia National Laboratories

U.S. DEPARTMENT OF
ENERGY

PUBLICATION THESIS OPTION

This thesis has been prepared in two styles utilized by two conferences. Pages 3-23 were submitted for publication to the Sixteenth International Workshop on Learning Classifier Systems using the ACM SIG format. Pages 24-45 were submitted for publication to the Symposium on Search-Based Software Engineering using the Springer LNCS format.

ABSTRACT

As computers are increasingly relied upon to perform tasks of increasing complexity affecting many aspects of society, it is imperative that the underlying computational methods performing the tasks have high performance in terms of effectiveness and scalability. A common solution employed to perform such complex tasks are computational intelligence (CI) techniques. CI techniques use approaches influenced by nature to solve problems in which traditional modeling approaches fail due to impracticality, intractability, or mathematical ill-posedness.

While CI techniques can perform considerably better than traditional modeling approaches when solving complex problems, the scalability performance of a given CI technique alone is not always optimal. Hybridization is a popular process by which a better performing CI technique is created from the combination of multiple existing techniques in a logical manner. In the first paper in this thesis, a novel hybridization of two CI techniques, accuracy-based learning classifier systems (XCS) and cluster analysis, is presented that improves upon the efficiency and, in some cases, the effectiveness of XCS.

A number of tasks in software engineering are performed manually, such as defining expected output in model transformation testing. Especially since the number and size of projects that rely on tasks that must be performed manually, it is critical that automated approaches are employed to reduce or eliminate manual effort from these tasks in order to scale efficiently. The second paper in this thesis details a novel application of a CI technique, multi-objective simulated annealing, to the task of test case model generation to reduce the resulting effort required to manually update expected transformation output.

ACKNOWLEDGMENTS

There are a number of people who have helped me greatly along my path to completing this thesis to which I owe my undying gratitude. First and foremost, I would like to thank my advisor Dr. Daniel Tauritz. After expressing my interest in his research area, he gave me my first break as an undergraduate researcher. Over the years since, he has pushed me to my full potential in both coursework and research alike. His encouraging work ethic, technical expertise, and masterful guidance has allowed me to achieve my academic and professional successes.

Dr. Marouane Kessentini helped me immensely through his instruction, expert domain knowledge, and research ideas. His patience in helping me understand how to apply my technical abilities to an unfamiliar domain is incredible. Dr. Samuel Mulder has been a phenomenal mentor professionally and academically. His willingness and ability to convey his expert knowledge has contributed greatly to my understanding of the theory and applicatication of many areas in computer science. I would like to thank Sandia National Laboratories for providing my funding through their Critical Skills Master's Program that made my graduate studies possible thus far. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

I wish to thank my entire extended family for supporting me over the years, especially my wife, Charity, and siblings, Brad and Sarah. Lastly, and most importantly, I wish to thank my parents, Geri and Scott, for raising, caring, supporting, teaching, and loving me. Their countless sacrifices and affinity for computers have made me who I am today. To them, I dedicate this thesis.

TABLE OF CONTENTS

Page

SECTION

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

1. INTRODUCTION

As computers are increasingly relied upon to perform tasks of increasing complexity affecting many aspects of society, it is imperative that the underlying computational methods performing the tasks have high performance in terms of effectiveness and scalability. A common solution employed to perform such complex tasks are computational intelligence (CI) techniques. CI techniques use approaches influenced by nature to solve problems in which traditional modeling approaches fail due to impracticality, intractability, or mathematical ill-posedness.

While CI techniques have been developed that perform well on artificial benchmark problems as well as relatively small real-world problems, their functionality does not always scale well when applied to larger and more complex artificial and real-world problems. One approach used to improve the functional scalability of CI techniques is a process known as hybridization. Hybridization is a popular process by which a better performing CI technique is created from the combination of multiple existing techniques in a logical manner. The first paper introduces a novel extension of the accuracy-based learning classifier system (XCS) termed Local Expert Accuracy-Based Learning Classifier System (LEXCS) that employs an automated problem decomposition technique to transform a problem into a number of simpler, disjoint subproblems based on the problem's spatial structure. Results are presented demonstrating LEXCS' improved scalability in terms of efficiency and, in some cases, classification accuracy in real-world classification problems and a novel artificial benchmark problem of varying size and difficulty. Furthermore, the effectiveness of employing a novel classifier migration technique is described.

A number of tasks in software engineering are performed manually, such as defining expected output in model transformation testing. Especially since the number of size of projects that rely on tasks that must be performed manually, it is critical that automated approaches are employed to reduce or eliminate manual effort from these tasks in order to scale efficiently. In the second paper, an effective and scalable approach to model transformation testing is proposed by refactoring the existing test case models, employed to test previous metamodel versions, to cover new changes. To this end, a multi-objective optimization algorithm is employed to generate test case models that maximizes the coverage of the new metamodel while minimizing the number of refactorings as well as test case model elements that have become invalid due to the new changes. Validation results on a widely used transformation mechanism confirm the effectiveness of the approach.

PAPER

I. Improving XCS Scalability with Automatic Problem Decomposition

Jeffery S. Shelburg[1], Daniel R. Tauritz[1], and Samuel A. Mulder[2]

[1]*Natural Computation Laboratory, Department of Computer Science, Missouri University of Science & Technology, Rolla, Missouri, U.S.A.*

[2]*Sandia National Laboratories, Albuquerque, New Mexico, U.S.A.*

ABSTRACT

XCS is a widely popular variant of LCS that creates a population of classifiers whose fitnesses are based upon their prediction accuracy. Although it performs sufficiently well on a variety of problems, there is still plenty of room for improvement in terms of scalability. This paper introduces a novel extension of XCS termed Local Expert Accuracy-Based Learning Classifier System (LEXCS) that employs an automated problem decomposition technique to transform a problem into a number of simpler, disjoint subproblems based on the problem's spatial structure. Results are presented demonstrating LEXCS' improved scalability in terms of efficiency and, in some cases, accuracy in real-world classification problems and a novel artificial benchmark problem of varying size and difficulty. Furthermore, the effectiveness of employing a novel classifier migration technique is described.

1. INTRODUCTION

Real-world problems and control processes are increasingly being offloaded to computers. These problems and processes are only getting larger and more difficult, so it is imperative that their underlying solutions perform well, be efficienct, and have the ability to scale well. While XCS performs sufficiently well over a wide variety of problems in general, its performance and efficiency can degrade quickly as the size and difficulty of problems increase. Specialized measures can be taken to overcome this scalability deficiency, but it normally requires problem knowledge and can be difficult and tedious to implement.

LEXCS, the XCS extension proposed in this paper, aims to alleviate XCS' scalability deficiency without requiring any prior problem knowledge through the use of automatic problem decomposition. A clustering algorithm and quality metric are used to automatically determine how a given problem space can be logically partitioned based on spatial structure so that one large problem is transformed into a number of disjoint, simpler problems that can be more easily approached. Additionally, a classifier migration technique utilizing a novel classifier mapping operator and donor subpopulation selection technique are proposed. Experiments on real datasets and an artficial benchmark are performed using LEXCS with classifier migration, LEXCS without classifier migration, and XCS to explore LEXCS' overall scalability performance as well as the effectiveness of the proposed classifier migration technique.

2. RELATED WORK

In [1], Gershoff and Schulenburg introduce Collective Behavior Hierarchical XCS (CB-HXCS) that takes a partitioned problem space and creates a hierarchy architecture of XCS instances where the bottom level XCS instances (micro agents) are assigned to partitions. Signals are sent up the hierarchy through multiple XCS

instances to the meta XCS agent that makes the final classification. Although CB-HXCS's classification performance on a multiplexer problem was shown to be an improvement over XCS, it requires multiple XCS instances at different levels to process input signals to produce a final classification signal. Because of this, CB-HXCS may require more classifier evaluations, and therefore more time and computational resources, than XCS to attain the same level of classification accuracy.

In [2], Richter et al. investigate different distributed XCS learning approaches on a novel simulation environment. Experimental results showed that manually decomposing the problem into emergence values and assigning separate XCS instances to respond to each value improved the speedup and classification performance over XCS. As with CB-HXCS, these learning approaches do not perform the problem space partitioning automatically as the authors manually partition problems using problem knowledge. Since problem knowledge is not always known a priori and manually partitioning a problem can be very tedious and difficult even with problem knowledge, this is not always a reliable method of problem space partitioning.

The research presented in [3] introduces Coevolutionary XCS (CoXCS) which randomly partitions the input feature space and assigns an XCS instance to each partition in order to induce restrictive mating resulting in feature subspace specialized classifiers. After a number of training iterations, a classifier migration episode occurs where randomly selected classifiers from subpopulations are migrated in a random topology to faciliate the sharing of specialized genetic material. Results from these experiments and subsequent experiments detailed in [4] not only show that CoXCS can outperform XCS in terms of classification accuracy, but that allowing classifiers to migrate between subpopulations yielded better results that disallowing it in these cases. In [5], Bull et al. show that adding classifier migration episodes to an LCS ensemble can reduce the number of training iterations needed to achieve optimal classification accuracy on multiplexer problems. Because classifier migration has been

shown to be beneficial in similar techniques that utilize multiple classifier populations, its effectiveness when used with LEXCS is explored in this paper.

In [6], the authors introduce a method of initializing an LCS ruleset using information extracted from the problem space using k-means clustering. In addition to improving classification accuracy and ruleset interpretability, this approach also achieved faster training speeds and smaller ruleset size. Since k-means clustering is powerful enough to successfully extract useful information from problem spaces while also maintaining a low time complexity, its use in problem decomposition is investigated with LEXCS.

In [7], it is noted that crossover operators that blindly recombine solution genes can disrupt complex solution structures. To combat this, specialized crossover operators can be created that preserve values of logically connected genes, but these can be complex and tedious to implement. In island models, such crossover operators are less likely to cause disruption since solutions in the same island tend to become similar to one another. Thus, using crossover in an island model as opposed to a global model can increase its success rate of creating fit offspring. Because LEXCS utilizes an island model where classifier subpopulations are isolated in terms of reproduction, it is believed that the use of common crossover operators in LEXCS will perform well even if the operators were disruptive when used globally on a problem in XCS. If true, the use of LEXCS in place of XCS could eliminate the need to obtain problem knowledge or implement specialized crossover operators for such problems.

Other methods used to improve XCS scalability include compaction algorithms such as Wilson's compact ruleset algorithm (CRA) [8] and a speedier alternate version called CRA2 introduced by Dixon et al. [9]. XCS variants such as XCS with code fragmented action (XCSCFA) [10] introduced by Iqbal et al. extract and use building blocks of problem domain knowledge to facilitate complex problem learning. Scalability improvement methods such as these are implemented at the population

and classifier action level while LEXCS is implemented at the problem domain level. As a result, LEXCS is not meant to replace such methods, but rather offer another level at which to potentially improve XCS scalability.

3. METHODOLOGY

3.1. LEXCS Algorithm. LEXCS is a simple extension of XCS that does not modify the core XCS algorithm; rather, LEXCS facilitates the automatic decomposition of problems as well as provides the ability for isolated rulesets to share rules This is accomplished in LEXCS by a priori decomposing the original problem into a number of simpler subproblems. The maximum population size (sum of classifier numerosities in XCS) is distributed amongst the partition classifier subpopulations based on the ratio of number of training objectsassigned to a given partition and the total number of all training objects. This is done to ensure each partition ruleset gets a fair share of the overall maxmimum population size. During each iteration of the training phase, an input is randomly drawn from the training set. The classifier subpopulation for the partition to which the drawn input belongs is then activated for use in the XCS algorithm. After XCS trains with the drawn input and activated classifier subpopulation, classifier migration between subpopulations is done if classifier migration criteria is met, in this case, number of training iterations since the last migration episode. The pseudocode for LEXCS with classifier migration can be found in Algorithm 1.

3.2. Automatic Problem Decomposition. To improve the scalability of XCS functionality, LEXCS first automatically decomposes a problem into smaller subproblems. A clustering algorithm and cluster validity measure are used for this task because of its ability to exploit the spatial layout of a given problem space to find suitable partitions that logically partitions a problem into subproblems without

Algorithm 1 LEXCS with Classifier Migration Pseudocode

```
Automatically partition training data
Initialize subpopulation for each partition
Allot maximum population size amongst partitions
while termination criteria is not met do
    Select random training input
    Activate subpopulation from input's partition
    Perform XCS algorithm
    if Classifier migration criteria is met then
        Perform classifier migration
```

any external problem knowledge in an automated fashion. This is done by first using the clustering algorithm to partition the training dataset into two partitions. The quality of the partition is then evaluated using the cluster validity measure. The number of partitions is then incremented by one followed by a new partitioning of the training dataset and quality measure until a decrease in partition quality from the previous partitioning is found. By decomposing the problem in this manner, neither the number of partitions to use nor the maximum number of clusters to evaluate need to be known a priori thus automating the problem decomposition process.

While there exist many clustering algorithms and cluster validity measures that can be used for automatic problem decomposition, the k-means clustering algorithm [11, 12] and Caliński and Harabasz index (CH index) [13] were chosen for use in LEXCS. k-means was chosen because it it maintains a low time complexity (approximately linear) and is therefore a good choice for use with large-scale datasets [14]. As a tradeoff, k-means suffers from the disadvantage of being prone to getting stuck in a local optimum because of its hill climbing optimization approach. For the task of logically partitioning a problem into smaller subproblems, this disadvantage is acceptable because partitioning is used to group similar items in the training set at a high level while the task of fine-grained, local expert classification within each partition is taken care of by the XCS algorithm within the training loop of LEXCS. The CH index

was the cluster validity measure chosen to evaluate quality of partitioning in LEXCS because it has been shown to be among the best performing measures [14, 15].

The k-means clustering algorithm is among the most popular and has a relatively simple implementation. Given a number of clusters k into which inputs must be clustered, k cluster centroids are randomly generated within the input space. Next, each input is assigned to the cluster whose centroid minimizes the Euclidean distance between the input and centroid. Then, all cluster centroids are recalculated by taking the arithmetic mean of the inputs assigned to the cluster over all dimensions. The process of assigning inputs to the nearest cluster centroids and recalculating the cluster centroids is repeated until the input cluster assignments do not change from the previous iteration. In LEXCS, only inputs from the training set are utilized during this process. The k-means pseudocode is shown in Algorithm 2.

Algorithm 2 k-means Clustering Pseudocode

```
Generate k random centroids within input space
repeat
    Assign each input to nearest cluster centroid
    Recalculate cluster centroids
until Cluster assignments do not change
```

The CH index measures the quality of a clustering structure based on cluster compactness, spread between clusters, number of inputs, and the number of clusters such that a higher quality clustering structure translates to a higher CH index value using Equation 1. $\mathbf{S_B}$ represents the scatter matrix of error sum of squares between different clusters (inter-cluster) while $\mathbf{S_W}$ represents the scatter matrix of squared differences of inputs from the centroids of their assigned cluster (intra-cluster). Furthermore, $\mathrm{Tr}(\mathbf{S_B})$ measures cluster compactness while $\mathrm{Tr}(\mathbf{S_W})$ measures the spread between clusters given that the notation $\mathrm{Tr}(\cdot)$ represents the linear algebra operation *trace* that yields the sum of the elements in the main diagonal of a matrix.

As shown in reduced form in Equation 2, Tr($\mathbf{S_B}$) is calculated by summing up the distance between the cluster centroid and the centroid of the all inputs multiplied by the number of inputs assigned to the cluster for each cluster. Tr($\mathbf{S_B}$) is initially a relatively large value (not very compact) when using two clusters and will decrease as the number of clusters increases until cluster compactness eventually decreases due to over-clustering. In its reduced form shown in Equation 3, Tr($\mathbf{S_W}$) is calculated by summing up the distance between an input and the centroid of its assigned cluster for all inputs. In constrast to the behavior of Tr($\mathbf{S_B}$), Tr($\mathbf{S_W}$) is initially a relatively small value (low cluster spread) when using two clusters and will increase as the number of clusters increases. When these are combined with the number of inputs, N, and number of clusters, K, to yield the CH index as defined in Equation 1, the resulting CH index is initially relatively small value using two clusters and will increase as the number of clusters increases until over-clustering occurs (stopping criterion).

$$CH(K) = \frac{Tr(\mathbf{S_B})}{K-1} \Big/ \frac{Tr(\mathbf{S_W})}{N-K} \tag{1}$$

$$Tr(\mathbf{S_B}) = \sum_{k=1}^{K} |C_k| \, \|\overline{C_k} - \bar{x}\|^2 \tag{2}$$

$$Tr(\mathbf{S_W}) = \sum_{k=1}^{K} \sum_{i=1}^{|C_k|} \|x_{k,i} - \overline{C_k}\|^2 \tag{3}$$

3.3. Classifier Migration. In LEXCS, classifier migration episodes take place after a number of training iterations where a percentage of a subpopulation's classifiers are migrated to another subpopulation as done in previously published work [3, 5]. However, since partitioning is done automatically based on the arbitrary spatial structure of the problem space and only one XCS instance is used to determine LEXCS's overall classification signal, a few changes had to be made. During a

migration episode, the subpopulation that will receive the immigrant classifiers is the one utilized in the most recent training iteration. By choosing the receiving subpopulation in this way, partitions that cover the subspaces containing the most inputs will receive the most genetic material from other subpopulations on average.

Next, the donor partition containing the subpopulation from which the immigrant classifiers will originate is chosen. Based on the premise that classifiers in partitions closest to the receiving partition are more likely to be relevant than those classifiers in partitions furthest away, LEXCS chooses the donor partition based on a probability distribution where the partitions closest to the receiving partition are given the highest probabilities of being chosen while those furthest away have the lowest. This probability that a given partition j is chosen as the donor partition given that partition i is the receiving partition is calculated based on the distance between their centroids using the soft-max equation as shown in Equation 4 [16]. After the donor partition is chosen, a percentage of its classifier subpopulation is chosen for migration using fitness proportional selection without replacement where a classifier's accuracy is used as fitness.

Before the immigrant classifiers are inserted into the receiving partition's subpopulation, a mapping operator must first be applied. Since each partition subpopulation is used in non-overlapping subspaces of the input space, immigrant classifiers that migrate from one partition to another may never match any inputs that exist within the receiving partitions subspace. This can happen if the subspace covered by the immigrant classifiers' conditions lies completely outside of the receiving partition's subspace. To overcome this limitation, the mapping operator checks if each feature of the immigrant classifiers match any training inputs of the receiving partition. If a feature is matched by at least one training input in the receiving partition, then that feature is not modified. If, however, a feature is not matched by any training input in the receiving partition, that immigrant classifier's feature is changed to a

"don't care" so that it will match. Mapping immigrant classifiers in this way maximizes the preservation of classifier genes while ensuring that immigrant classifiers are relevant to the receiving partition's subspace and maximally general.

$$p_{ij} = \begin{cases} \frac{e^{-||x_i - x_j||^2}}{\sum_k e^{-||x_i - x_k||^2}} & \text{if } j \neq i \\ 0 & \text{if } j = i \end{cases} \tag{4}$$

4. EXPERIMENTAL SETUP

To determine the effectiveness of the proposed automatic problem decomposition and classifier migration strategy, experiments were carried out for XCS, LEXCS with classifier migration, and LEXCS without classifier migration. The quality of results is determined by two metrics: classification accuracy and number of classifiers evaluated. The number of classifiers evaluated metric is used instead of number of training iterations or wall time because neither metrics are sufficient for accurate speed comparisons. Not all training iterations take the same amount of time to process, especially with multiple classifier subpopulations, because the number of classifiers evaluated in any given training iteration are not the same. Using wall time measurements can be problematic especially on multiuser computers because other processes running concurrently can greatly affect resulting runtimes. Furthermore, wall time is heavily dependent on the software implementation as well as computer hardware specifications, so it is very difficult for other researchers to compare against. Using the number of classifiers evaluated during training iterations is an accurate metric because all classifier evaluations have the same constant computational cost and it is not affected by concurrently running processes, implementation, or computer hardware specifications so other researchers can easily compare against the metric.

The real-valued XCS (XCSR) [17] variant of XCS is used in the experiments carried out in this paper because real-valued features appear very often in real-world problems and they make it easier to create and control the difficulty of arbitrary spatial structures in artificial benchmark problems. The experiments were performed on problems based on two types of datasets: real-world datasets and novel artificial datasets. The three real-world datasets used are the diabetes, blood transfusion, and ecoli datasets from the UC Irvine Machine Learning Repository [18]. To show how performance scales with the size of the dataset, experiments were done on randomly sampled versions of these datasets at 25%, 50%, 75%, 100% for a total of 12 real-world datasets.

The novel artificial datasets introduced in this paper called Fuzzy Hyperspheres (FHS) are composed of two features where inputs of the same class exist in clusters arranged in a square grid formation as shown in Figure 1. The clusters are created by a Gaussian distribution about evenly spaced cluster centroids. These datasets vary by two measures: number of class clusters (affecting the size of the dataset and number of classes) and the fuzziness of the classification boundaries between the class clusters (affecting classification difficulty). The fuzziness is varied based on how many standard deviations of the underlying Gaussian distributions the Manhattan neighbor class cluster centroids are from each other. The number of class clusters is varied between 4, 9, 16, and 25 while the number of standard deviations between neighboring class cluster centroids is varied between 3.5, 3.75, 4.0, 4.25, and 4.5 to make a total of 20 different FHS datasets. The purpose of the FHS datasets is to showcase how a traditional crossover operator (uniform crossover) can be disruptive when used with XCS while performing well with LEXCS. This is due to the fact that a crossover of two randomly selected inputs in the entire problem space will likely yield an offspring somewhere between the two parents where inputs of neither of the parents' classes exist. The same operator will likely perform well when the

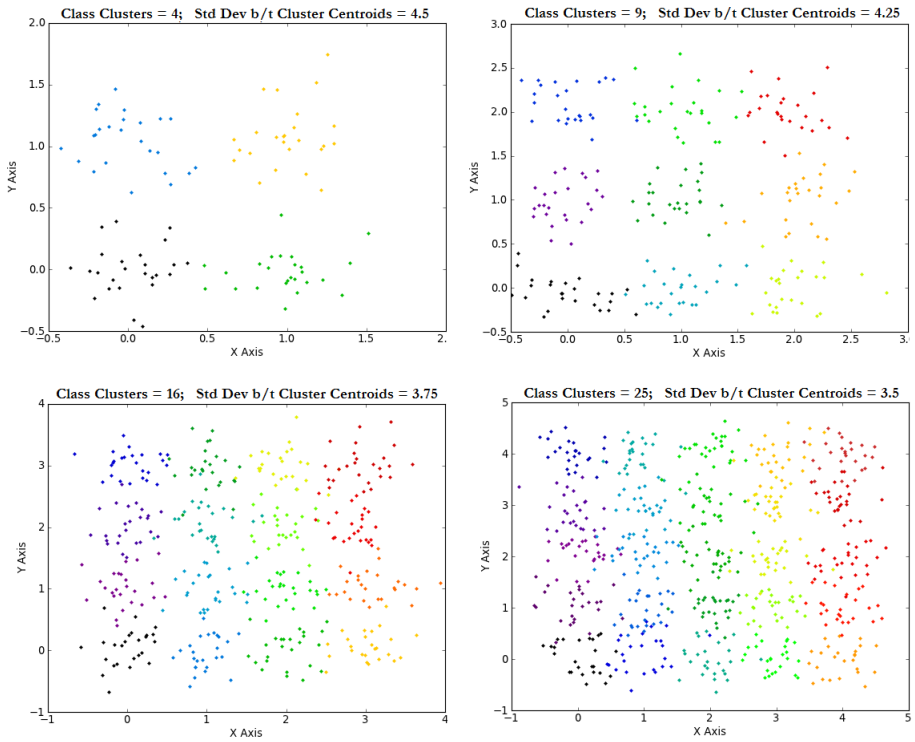

Figure 1: FHS datasets increasing in size, classes, and classification difficulty

problem space is logically partitioned based on spatial structure because an offspring of two parents from the same problem subspace is more likely to be relevant. Detailed descriptions of all of these datasets can be found in Table 1.

To ensure that parameter configurations are not unfairly skewed to induce more favorable results from either XCS or LEXCS, a wide variety of parameter configurations are tested for each. The commonly used XCS parameter values shown in Table 2 were taken from Wilson's algorthmic description of XCS [19] while the remaining XCS parameter settings are shown in Table 3. The crossover and mutation rates were chosen within the suggested ranges from [19] with the exception of the 1/#Features mutation rate which was found to yield good results in preliminary testing. The iterations between migration episodes and percentage of classifier subpopulation to migrate parameter settings were derived from suggested ranges given

Table 1: Dataset details

Dataset	**Instances**	**Features**	**Majority**
Diabetes	768	8	65%
Transfusion	748	4	76%
Ecoli	336	7	43%
FHS_4	100	2	25%
FHS_9	225	2	11%
FHS_{16}	400	2	6%
FHS_{25}	625	2	4%

Table 2: Commonly used XCS parameter settings

Parameter	**Value**
Learning Rate	0.15
GA Threshold	37
Deletion Threshold	20
Subsumption Threshold	20
Exploration Probability	0.5
Min Represented Actions	#Actions
DoGASubsumption	True
DoActionSetSubsumption	True
α	0.1
ϵ_0	0.1
v	5
δ	0.1
p_I	0.001
ϵ_I	0.001
F_I	0.001

in [5] and the mutation range parameter for XCSR was taken from [17]. The remaining parameter settings were values that were found to yield good performance in preliminary testing. All experiments were performed with three runs of ten-fold stratified cross validation for a total of 30 test folds in order to establish a baseline for statistical significance.

Table 3: XCS parameter settings

Parameter	**Values**
Crossover Rates	55%, 75%, 95%
Mutation Rates	1%, 5%, 1/#Features
Iterations b/t Migrations	100, 500, 1000, ∞
Percent Pop. to Migrate	1%, 5%, 10%
Converge Iters w/o Improve	5000
Max Pop. (Numerosity)	#Inputs $\times$ #Classes
Crossover Operator	Uniform
Mutation Range	$\pm$10%

5. RESULTS

For each dataset, only the results from the parameter configurations for LEXCS with rule migration, LEXCS without rule migration, and XCS with the highest average accuracy are used for comparison. Detailed results for the experiments can be found in Tables 4 - 7 where the first value in each cell represents the average over all 30 folds and the second value in parenthesis represents the standard deviation. Graphs depicting traces of accuracy versus number of classifiers evaluated for the experiments are shown in Figures 2 - 5. Note that the x-axis (number of classifiers evaluated) is logarithmically scaled and trace lines end at the number of classifiers evaluated when training is terminated due to convergence. The differences in accuracy for each sampling size of the three real-world datasets are statistically insignificant using an independent two-sample t-test except where noted. Furthermore, all differences in the number of classifiers evaluated between the LEXCS configurations with and without classifier migration are statistically insignificant while all differences between these results for the LEXCS configurations and the XCS configurations are statistically significant using the aforementioned test.

For the sake of succinctness, only the results of four FHS dataset configurations are included in the FHS graphs and table. To obtain the best coverage of the FHS

Table 4: Ecoli dataset results

Data	Config	% Acc.	Classifiers Eval'd	Pop. Size	Partitions
25%	CM	87.2 (10.3)	4.0e+04 (2.1e+04)	7.8 (12.6)	13.5 (2.1)
	No CM	85.8 (11.0)	3.8e+04 (2.1e+04)	7.4 (9.5)	13.6 (2.1)
	XCS	89.2 (8.2)	2.3e+05 (1.3e+05)	49.2 (1.9)	1.0 (0.0)
50%	CM	83.4 (8.3)	9.9e+04 (2.2e+04)	13.7 (15.6)	15.2 (2.4)
	No CM	85.2 (9.6)	9.4e+04 (3.7e+04)	13.5 (15.6)	15.4 (2.7)
	XCS	84.0 (6.6)	1.1e+06 (3.4e+05)	131.8 (4.0)	1.0 (0.0)
75%	CM	83.5 (7.2)	9.7e+04 (2.7e+04)	13.6 (15.7)	20.0 (3.6)
	No CM	82.7 (7.9)	1.0e+05 (2.5e+04)	13.5 (14.6)	20.0 (3.1)
	XCS	83.2 (3.8)	1.4e+06 (3.4e+05)	179.5 (6.4)	1.0 (0.0)
100%	CM	84.4 (6.8)	1.2e+05 (3.4e+04)	16.6 (12.9)	21.6 (3.3)
	No CM	84.1 (7.6)	1.2e+05 (3.6e+04)	15.9 (12.6)	22.5 (4.0)
	XCS	82.6 (5.1)	2.5e+06 (9.8e+05)	259.4 (6.6)	1.0 (0.0)

datasets, the size and complexity are changed simultaneously such that the four FHS datasets range from a small size with a low number of classes and simple classification task to a large dataset with a high number of classes and difficult classification task. For all FHS dataset results, the difference in LEXCS configuration results are statistically insignificant while the difference between LEXCS configurations and XCS are all statistically significant. As shown in Figure 6, XCS evaluates many times more classifiers than LEXCS as the sample size and classification difficulty of the datasets increase.

6. DISCUSSION

For the real-world dataset experiments, LEXCS attained classification accuracies that are statistically indistinguishable from or statistically better than the corresponding accuracies attained by XCS for all but two dataset configurations (the 75% and 100% sample of the Diabetes dataset, although differences were <3%). However, LEXCS attained these accuracies evaluating approximately one-tenth or fewer

Table 5: Diabetes dataset results

Data	Config	% Acc.	Classifiers Eval'd	Pop. Size	Partitions
25%	CM	86.3 (4.9)	2.4e+04 (5.5e+03)	3.6 (10.3)	18.7 (3.9)
	No CM	86.5 (4.7)	2.5e+04 (6.7e+03)	3.5 (8.1)	18.5 (3.3)
	XCS	88.3 (3.9)	2.6e+05 (5.8e+04)	36.6 (1.2)	1.0 (0.0)
50%	CM	85.7 (4.3)	3.6e+04 (1.1e+04)	4.3 (7.4)	25.4 (4.6)
	No CM	84.7 (4.0)	3.5e+04 (1.2e+04)	4.7 (9.6)	24.2 (4.3)
	XCS	86.8 (3.9)	6.1e+05 (1.7e+05)	76.0 (0.3)	1.0 (0.0)
75%	CM	81.6 (4.5)	4.0e+04 (1.5e+04)	5.3 (6.5)	28.4 (5.7)
	No CM	81.1 (4.9)[1]	4.4e+04 (1.4e+04)	5.6 (7.2)	27.6 (4.6)
	XCS	83.9 (4.3)[1]	1.0e+06 (2.8e+05)	113.6 (0.9)	1.0 (0.0)
100%	CM	80.5 (2.7)[1]	5.1e+04 (1.9e+04)	6.0 (8.0)	32.4 (6.2)
	No CM	80.8 (3.0)[1]	5.2e+04 (1.7e+04)	5.9 (6.0)	32.5 (3.8)
	XCS	83.2 (2.1)[1]	1.3e+06 (4.1e+05)	151.5 (0.8)	1.0 (0.0)

Table 6: Transfusion dataset results

Data	Config	% Acc.	Classifiers Eval'd	Pop. Size	Partitions
25%	CM	90.2 (4.0)[1]	1.7e+04 (5.0e+03)	2.7 (3.0)	17.4 (2.2)
	No CM	91.1 (4.4)[1]	2.0e+04 (6.9e+03)	2.9 (3.8)	16.9 (2.6)
	XCS	87.6 (3.7)[1]	2.3e+05 (5.6e+04)	35.7 (1.1)	1.0 (0.0)
50%	CM	86.5 (4.5)[1]	3.7e+04 (1.3e+04)	5.2 (2.6)	17.7 (3.2)
	No CM	86.5 (3.4)[1]	3.6e+04 (1.1e+04)	4.9 (2.7)	17.7 (1.7)
	XCS	84.0 (3.2)[1]	5.0e+05 (1.8e+05)	71.7 (0.9)	1.0 (0.0)
75%	CM	83.6 (2.5)	4.8e+04 (1.3e+04)	6.6 (2.6)	21.0 (2.0)
	No CM	83.3 (2.5)	5.2e+04 (1.6e+04)	6.5 (2.9)	21.3 (2.7)
	XCS	82.8 (2.4)	8.3e+05 (3.1e+05)	108.6 (1.6)	1.0 (0.0)
100%	CM	83.9 (2.1)[1]	6.6e+04 (1.8e+04)	8.9 (3.0)	21.8 (2.7)
	No CM	83.8 (1.9)[1]	6.4e+04 (2.0e+04)	8.9 (2.7)	21.6 (3.1)
	XCS	82.7 (2.1)[1]	1.1e+06 (3.1e+05)	144.3 (3.7)	1.0 (0.0)

[1] Difference in results is statistically significant using an independent two-sample t-test with α=0.05

Table 7: FHS dataset results

CL,SD	Config	% Acc.	Classifiers Eval'd	Pop. Size	Partns
4,4.5	CM	100.0 (0.0)	1.4e+03 (2.2e+03)	4.1 (9.6)	14.4 (2.6)
	No CM	100.0 (0.0)	2.2e+03 (4.1e+03)	4.2 (8.0)	13.8 (2.2)
	XCS	93.8 (9.5)	1.0e+05 (6.5e+04)	31.8 (0.8)	1.0 (0.0)
9,4.25	CM	90.4 (6.9)	7.1e+04 (2.9e+04)	11.0 (12.1)	19.1 (1.9)
	No CM	90.0 (5.8)	7.3e+04 (2.5e+04)	11.1 (11.7)	18.9 (2.3)
	XCS	49.8 (6.2)	1.0e+06 (2.3e+05)	145.7 (9.0)	1.0 (0.0)
16,3.75	CM	83.3 (9.4)	1.8e+05 (4.6e+04)	24.4 (19.0)	23.7 (3.9)
	No CM	82.2 (9.6)	2.0e+05 (6.1e+04)	24.4 (16.9)	23.9 (4.1)
	XCS	29.0 (3.6)	2.6e+06 (5.7e+05)	397.7 (22.1)	1.0 (0.0)
25,3.5	CM	78.1 (5.4)	4.7e+05 (1.3e+05)	51.1 (8.3)	26.1 (3.2)
	No CM	78.1 (4.9)	4.7e+05 (9.8e+04)	50.5 (5.6)	26.2 (3.0)
	XCS	19.7 (2.4)	6.1e+06 (1.4e+06)	785.4 (51.8)	1.0 (0.0)

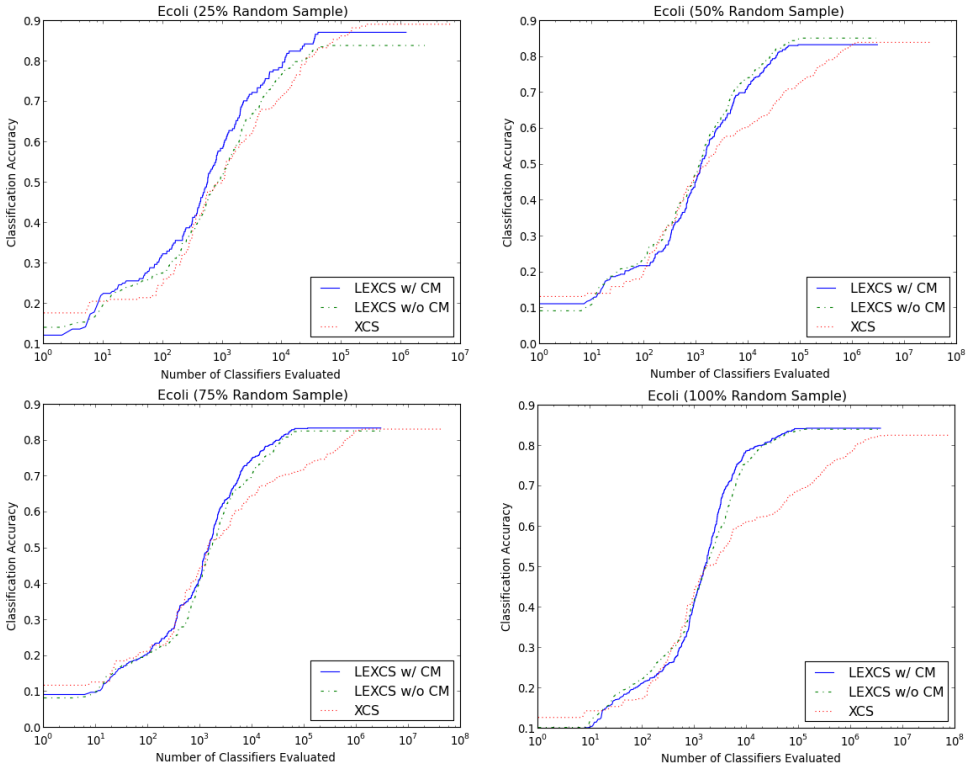

Figure 2: Results for the Ecoli dataset

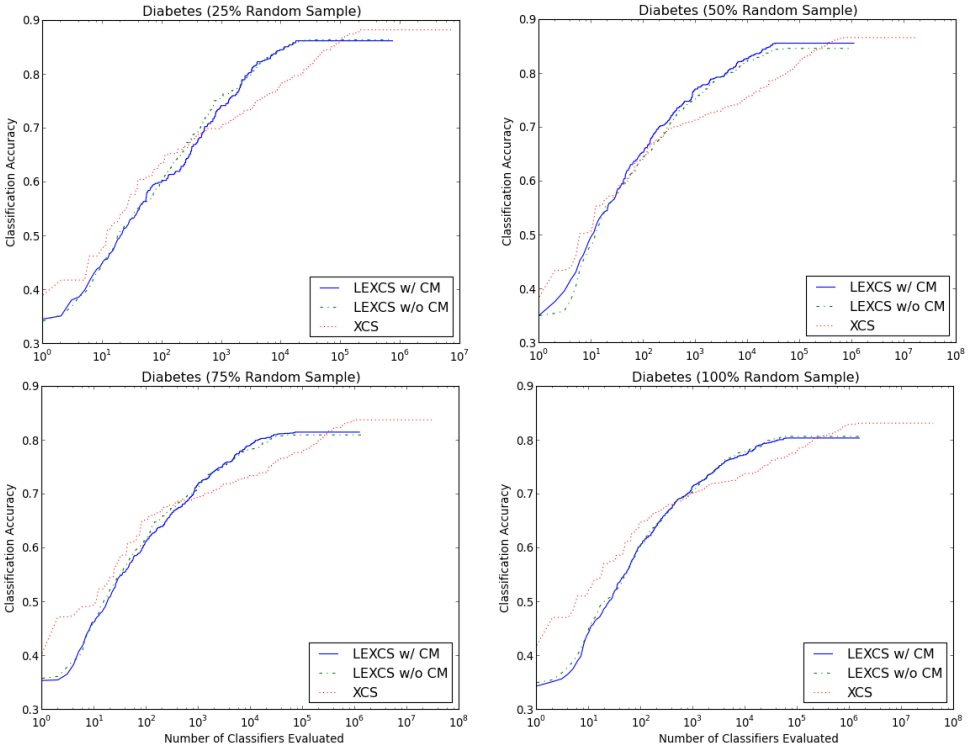

Figure 3: Results for the Diabetes dataset

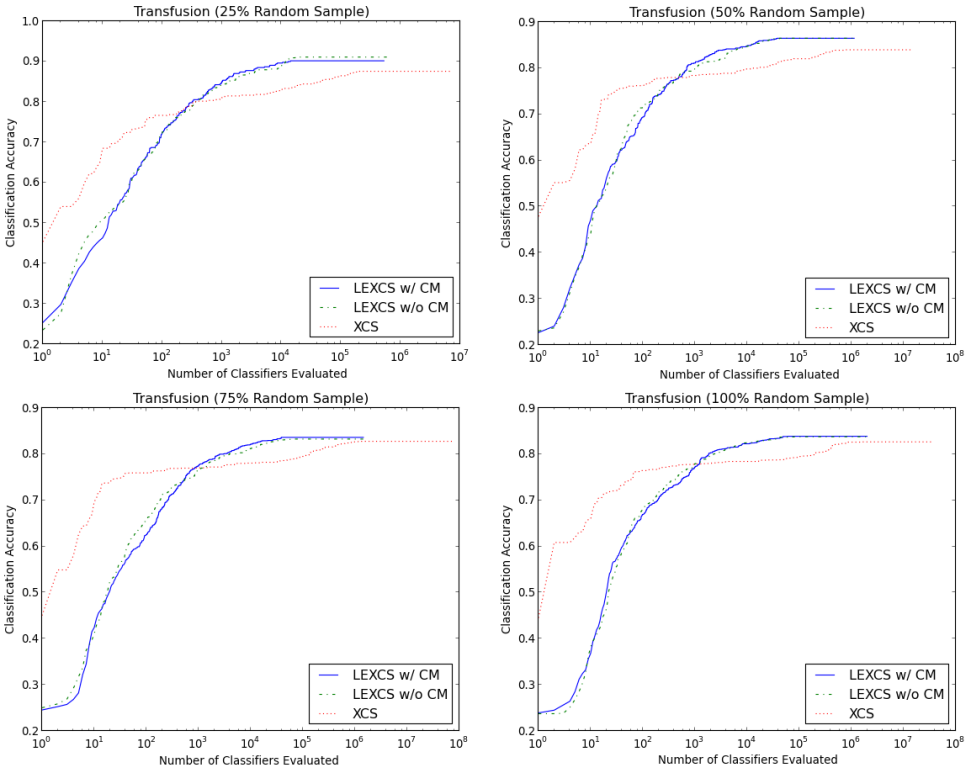

Figure 4: Results for the Transfusion dataset

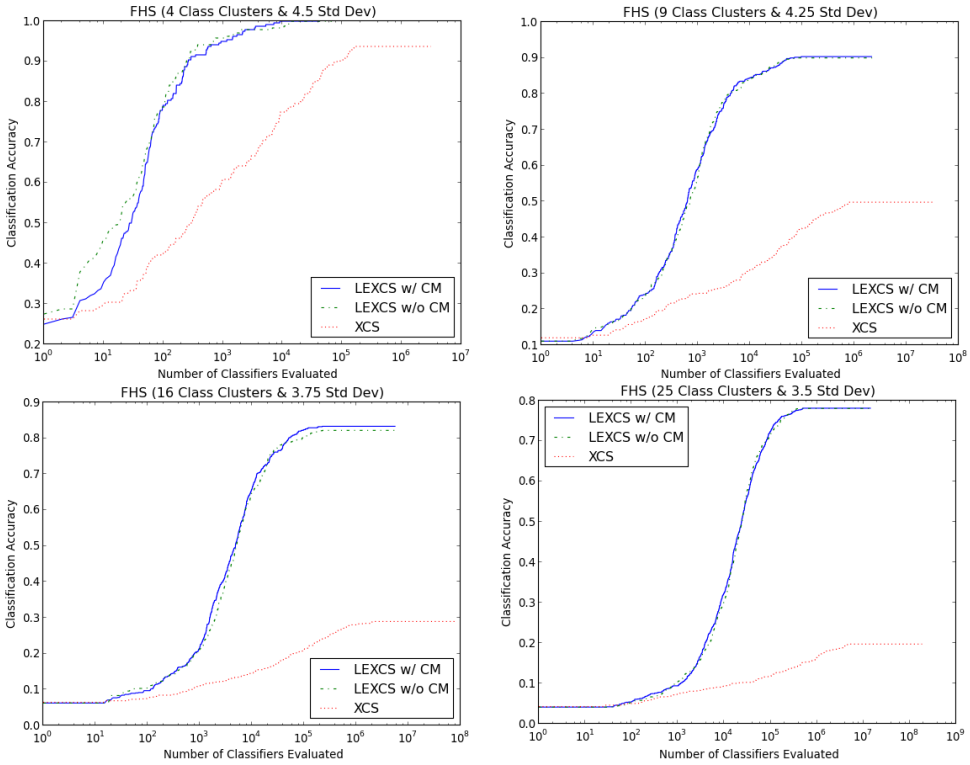

Figure 5: Results for the FHS datasets

classifiers than XCS, making LEXCS overall much more efficient in terms of accuracy versus number of classifiers evaluated.

In the experiments on the FHS datasets, the gap between the accuracies attained by LEXCS and XCS increased significantly as the number of class clusters and classification difficulty were increased as can be seen from the graphs in Figure 5. Similarly to the real-world dataset experiment results, FHS dataset experiment results show than LEXCS evaluated less than one-tenth the number of classifiers than XCS. These results show that LEXCS can overcome the downfalls of utilizing a common crossover operator an entire problem space that is disruptive without requiring the implementation of any specialized crossover operator. This is important because problem knowledge will not always be known a priori that could be used to formulate

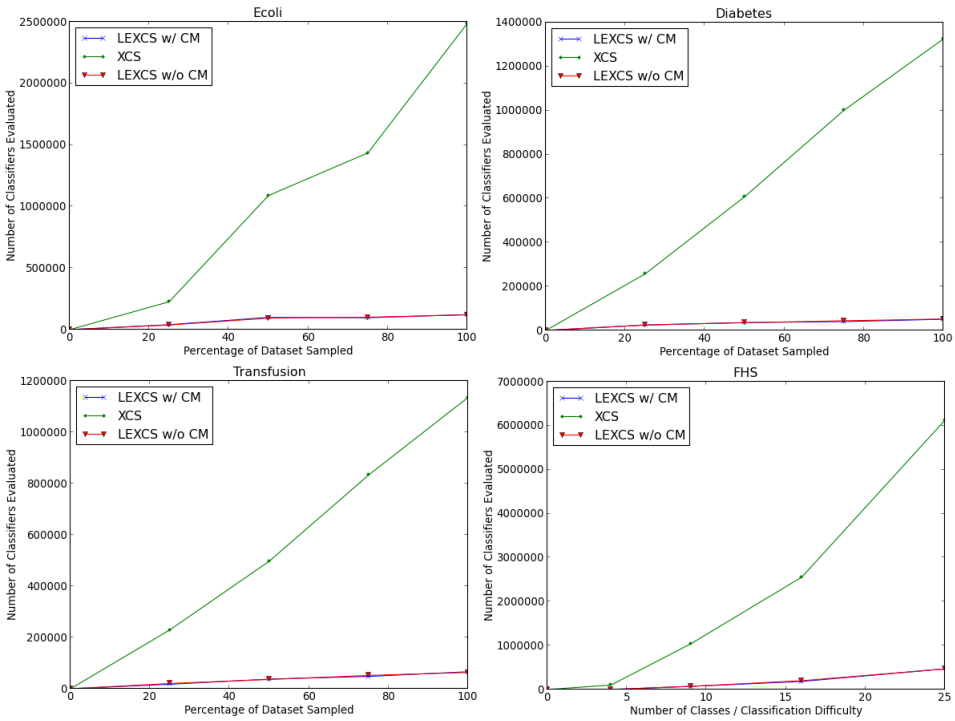

Figure 6: Number of classifiers evaluated for each dataset at each size. FHS results also varied from least to most difficult classification simultaneously with smallest to largest dataset size.

a specialized crossover operator and the implementation of such specialized crossover operators can be difficult and tedious.

In addition to training faster, the results obtained from these experiments suggest that LEXCS likely also has the advantage of creating a classification model that classifies new inputs more efficiently than XCS' resulting classification model. In the LEXCS model, a new input is first assigned to a partition using the nearest neighbor algorithm with the partition centroids and then classified using that partition's classifier subpopulation whereas the XCS model only uses its classifier population to perform classification. Although the total number of classifiers in the LEXCS model (sum of subpopulation sizes) is slightly more than the number of classifiers in the XCS population on average, the number of items evaluated to yield a classification using the LEXCS model (partition centroids + one partition's classifiers) is less than

that of the XCS model (all classifiers). Since the evaluation of partition centroids and classifiers both require comparing its features to the input's features, LEXCS should be able to classify new inputs quicker than XCS for all resulting models obtained in these experiments.

Results from the experiments performed indicate that there is no statistical difference in accuracy or number of classifiers evaluated between LEXCS with and without classifier migration. This could be due to partition subpopulations' ability to quickly and accurately map their assigned problem subspaces on their own. Another possibility is that the chosen datasets and the manner in which they were partitioned were such that classification patterns were largely localized within each partition's subspace so that performance improvements due to immigrant classifiers were limited at best.

7. CONCLUSION

On the real-world dataset problems used in experimentation, LEXCS performed approximately equivalent to XCS in terms of accuracy while the number of classifiers it evaluated were much less than that of XCS making it overall much more efficient and scalable. For the artificial benchmark dataset problems presented in this paper, LEXCS not only consistently evaluated many fewer classifiers than XCS, but its accuracy beyond that of XCS increased as problem size and difficulty increased. This suggests that the structure of the novel artificial datasets exemplifies a drawback of using a common crossover operator with XCS, while demonstrating LEXCS' superiority. Employing the proposed novel classifier migration technique in LEXCS did not yield statistically improved results.

II. Regression Testing for Model Transformations: A Multi-Objective Approach

Jeffery S. Shelburg, Marouane Kessentini, Daniel R. Tauritz

Department of Computer Science

Missouri University of Science & Technology

Rolla, Missouri, U.S.A.

ABSTRACT

In current model-driven engineering practices, metamodels are modified followed by an update of transformation rules. After this is done, the updated transformation mechanism should be validated to ensure quality and robustness. Model transformation testing is a recently proposed effective technique used to validate transformation mechanisms. In this paper, a more efficient approach to model transformation testing is proposed by refactoring the existing test case models, employed to test previous metamodels/rules versions, to cover new changes. To this end, a multi-objective optimization algorithm is employed to generate test case models that maximizes the coverage of the new metamodel while minimizing the number of refactorings as well as test case model elements that have become invalid due to the new changes. Validation results on a widely used transformation mechanism confirm the effectiveness of our approach.

1. INTRODUCTION

Model-Driven Engineering (MDE) considers models as first-class artifacts during the software lifecycle. The number of available tools, techniques, and approaches

for MDE are growing that support a huge variety of activities such as model creation, model transformation, and code generation. The use of different domain-specific modeling languages and diverse versions of the same language increases the need for interoperability between languages and their accompanying tools [20]. Therefore, metamodels are regularly modified/evolved and their respective transformation rules updated.

After evolving metamodels, the updated transformation mechanism should be validated to assure quality and robustness. One efficient validation method proposed recently is model transformation testing [20, 21] which consists of generating a large number of different source models as test cases, applying the transformation mechanism to them, and verifying the result using an oracle function such as a comparison with an expected result. Two challenges are: the efficient generation of test cases, and the definition of the oracle function. This paper focuses on the efficient generation of test cases.

The generation of test cases for model transformation mechanisms is challenging because many issues need to be addressed. As explained in [22], testing model transformation is distinct from testing traditional implementations: the input data are models that are complex when compared to simple data types which complicates the generation and evaluation of test cases [23]. The basis of the work presented in this paper starts from the observation that most existing approaches in testing evolved transformation mechanisms regenerate all test cases from scratch. However, this can be a very fastidious task since the expected output for all test cases needs to be redefined. A better strategy is to revise existing test cases to cover new changes in metamodels to reduce the effort required to redefine expected test case results.

In this paper, a multi-objective search-based approach is used to generate test case models that maximizes the coverage of the new metamodel while minimizing the number of refactorings and test case model elements that have become invalid due to

the new changes. The proposed algorithm is an adaptation of multi-objective simulated annealing (MOSA) [24] and aims to find a Pareto optimal solution consisting of test case model refactorings that will yield the new test case models when applied to the test case models of the previous version that best satisfy the three criteria previously mentioned.

This approach is implemented and evaluated on a known case of transforming UML 1.4 class diagrams to UML 2.0 class diagrams [25]. Results detailing the effectiveness of the proposed approach are compared to results of a traditional simulated annealing (SA) approach (whose single objective is to maximize metamodel coverage) to create UML 2.0 test case models in two scenarios: (1) updating test case models for UML 1.4 and (2) creating new test case models from scratch. The results indicate that the proposed approach has great promise: based on 30 runs for each approach, MOSA outperforms SA in terms of minimizing the number of refactorings while, however, underperforming the SA approach in terms of maximizing metamodel coverage in both scenarios. Since the number of invalid test case model elements is always zero for the scenario where test case models are created from scratch, the proposed approach outperforms the SA approach in terms of minimizing the number of invalid test case model elements only in the approach updating UML 1.4 test case models.

The primary contributions of this paper are summarized as follows: (1) The paper introduces a novel formulation of the model transformation testing problem using a multi-objective optimization technique, and to the best of our knowledge, this is the first paper in the literature to use this technique to test the evolution of metamodels and transformation mechanisms, and (2) the paper reports the results of an empirical study with an implementation of the proposed MOSA approach compared to a traditional SA approach. The obtained results provide evidence supporting the claim that MOSA is more efficient than SA and starting from existing test case models is more effective than regenerating all test case models from scratch.

2. METHODOLOGY

In this section, the three main components of any search-based approach are defined: the solution representation, change operators, and objective function.

2.1. Solution Representation. Since the proposed approach needs to modify test case models in response to changes at the metamodel level, the solution produced from MOSA should yield a modified version of the original test case models that best conforms to the updated metamodel. This can be done primarily in one of two different ways: the solution could either consist of the actual updated test case model itself, or represent a structure that, when applied to the original test case models, produces the updated test case models. The latter was chosen for this problem in the form of lists of model refactorings, because it allows MOSA to modify a sequence of refactorings out of order. Modifying the updated test case models directly is more restrictive, because changes can only be applied to the current state of the updated test case models in the search-based process, whereas modifying lists of model refactorings allows for modifications at any point in the sequence of refactorings, such as removing, modifying, or inserting refactorings anywhere within the lists. By allowing such modifications, MOSA is able to reach far more possible test case models with a single solution modification, because it has the ability to, for example, modify or remove suboptiomal refactorings as well as insert well-performing refactorings anywhere in a sequence of refactorings.

If test case models were modified directly, the resulting sequence of refactorings executed to transform the original test case model to the updated test case models would be very long and likely include suboptimal refactorings as refactorings could only be added and not removed or modified. Ensuring that the resulting list of refactorings is as small as possible and contains the least amount of suboptimal

refactorings is important in software engineering because it makes the task of updating the test case models less difficult, more effective, and easier to understand.

The lists of refactorings solution representation consists of a set of vectors whose elements are refactorings that are applied to their corresponding test case model in the order in which they appear in the vector, where each vector of refactorings corresponds to one test case model. After applying the refactorings, the test case models will be transformed into the updated test case models that better conform to the updated metamodel. Figure 1 shows an example of a possible list of refactorings for a test case model that moves method *getAge* from class *Employee* to class *Person*, adds a *Salary* field to the *Employee* class, and then removes the *Job* class, in that order.

MoveMethod(getAge, Employee, Person)	AddField(Salary, Employee)	RemoveClass(Job)

Figure 1: Example list of refactorings

2.2. Change Operators. The only change operator employed in MOSA is mutation. When mutating a given test case model's list of refactorings, the type of mutation to perform is first determined from a user-defined probability distribution that chooses between inserting a refactoring into the list, removing a refactoring from the list, or modifying a refactoring in the list. When inserting a refactoring into a list of refactorings, an insertion point between refactorings is first chosen, including either ends of the list. The refactorings that appear in the list before the insertion point are first applied to the test case model in the order in which they appear in the list. A refactoring is then randomly generated for the refactored test case model as it exists at the selection point, applied to the model, and inserted into the list at the insertion point. The refactorings that appear after the insertion point in the list are then validated in the order in which they appear by first checking their validity and subsequently applying them to the test case model if they are valid. If a refactoring

is found to be invalid due to a conflict caused by the insertion of the new refactoring into the list, the refactoring is removed from the list. An invalid refactoring could occur if, for example, a new refactoring is inserted into the beginning of a list of refactorings that removes a model element referenced by a refactoring that appears later in the list. When performing a mutation that removes a refactoring from a list of refactorings, a refactoring is selected at random and removed from the list. Validation is performed in the same manner as when inserting a refactoring for those refactorings that appear after the removed refactoring in the list of refactorings.

When mutating a refactoring in the list of refactorings, a refactoring is first randomly selected. Then, one of three types of mutations is performed on the selected refactoring from a user-defined probability distribution. The types are: to either replace the selected refactoring with a new randomly-generated refactoring, replace the selected refactoring with a new randomly-generated refactoring of the same refactoring type, or mutate a parameter of the selected refactoring. An example of a refactoring parameter mutation is changing the target class of a *MoveMethod* refactoring to another randomly chosen class in the model. Validation for all three types of refactoring mutations are performed in the same manner as described previously.

2.3. Objective Functions. Objective functions are a very important component of any search-based algorithm, because they define the metrics upon which solutions are compared that ultimately guides the search process. In the context of determining the quality of lists of refactorings to be applied to test case models in response to metamodel changes, three objective functions that define characteristics of a good solution are: (1) maximize target metamodel coverage, (2) minimize model elements that do not conform to the target metamodel, and (3) minimize the number of refactorings used to refactor the existing source models.

Maximizing the coverage of the target metamodel is imperative because the sole purpose of test case models is to ensure that the model transformation mechanisms are robust. Minimizing the number of invalid test case model elements due to metamodel changes, ensures that the test case models themselves are free of defects in order to properly assess the quality of the model transformation mechanism being tested. Finally, minimizing the number of refactorings used to refactor the test case models reduces the amount of effort required to update the expected output for the test case model transformations.

2.3.1. Metamodel coverage. The method used to derive metamodel coverage was first introduced in [23]. This method begins by a priori performing partition analysis in which the types of coverage criteria taken into consideration for a given problem are chosen. For metamodel coverage, an adaptation of the same three coverage criteria from [23] are used. These criteria are association-end multiplicities (AEM), class attributes (CA), and generalizations (GN). AEM refers to the types of multiplicities used in associations included in a metamodel such as *0..1*, *1..1*, or *1..N*. CA refers to the types of class attributes included in a metamodel such as *integer*, *string*, or *boolean*. Since both source and target metamodels used in the empirical tests in this paper support class operations in addition to attributes, class method return types are included in CA. GN refers to the coverage of classes that belong to each of the following categories: *superclass*, *subclass*, *both superclass and subclass*, and *neither superclass nor subclass*.

Each coverage criterion must be partitioned into logical partitions that, when unioned together, represent all the value types each criterion could take on. These partitions are then assigned representative values to represent each coverage criterion partition. For example, if a metamodel allows for classes to have an *integer* attribute, then the *integer* class attribute element is included in the CA coverage criterion. The values an *integer* class attribute can take on can be split into partitions whose

representative values are *<-1*, *-1*, *0*, *1*, and *>1*, for example. An example of partition analysis and a subset of the coverage items generated from its representative values are shown in Table 1 and Table 2, respectively.

Table 1: Partition analysis example

Coverage Criteria	**Representative Values**
CA: *boolean*	*true, false*
CA: *integer*	*<-1, -1, 0, 1, >1*
CA: *float*	*<-1.0, -1.0, 0.0, 1.0, >1.0*
CA: *string*	*Null, '', 'something'*
AEM: *1..1*	*1*
AEM: *1..N*	*1, N*
GN	*sub, super, both, neither*

Table 2: Example coverage items created from values in Table 1

Constraint 1	**Constraint 2**
CA: *-1*	AEM: *N*
CA: *'something'*	GN: *super*
AEM: *1*	AEM: *N*
AEM: *1*	GN: *neither*
CA: *false*	CA: *>1.0*
CA: *Null*	AEM: *1*

After representative values are defined, a set of coverage items for the target metamodel is created. In our adaptation of the coverage item set creation method introduced in [23], this is done by calculating all possible tuple combinations of representative values from all partitions of all coverage criteria types that are included in the target metamodel. The exception being the coverage items containing two different GN representative values, because they would be impossible to satisfy. The metamodel coverage objective value for given test case models and target metamodel is determined by calculating the percentage of metamodel coverage items the test case models satisfy. For example, if a given target metamodel included associations with end multiplicities of *1..1* $\rightarrow$ *1..N*, then the derived coverage items would include associations with end-multiplicities of *1* $\rightarrow$ *1* and *1* $\rightarrow$ *N*. Additionally, if a given

target metamodel also included *boolean* class attributes, then the additional coverage items would include classes with a *boolean* attribute and association end multiplicity of *true* and *1*, *false* and *1*, *true* and *N*, and *false* and *N*, respectively. For a more in-depth example of a model and the coverage items it would satisfy, refer to Figure 2 and Table 3, respectively.

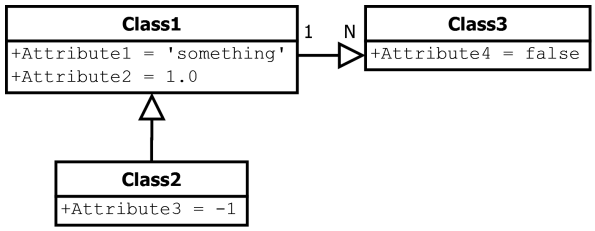

Figure 2: Example test case model

Table 3: Coverage items satisfied by the example shown in Figure 2

Constraint 1	**Constraint 2**
CA: *'something'*	CA: *1.0*
CA: *'something'*	AEM: *1*
CA: *1.0*	AEM: *1*
CA: *'something'*	GN: *super*
CA: *1.0*	GN: *super*
AEM: *1*	GN: *super*
CA: *-1*	GN: *sub*
CA: *false*	AEM: *N*
AEM: *1*	AEM: *N*
AEM: *N*	GN: *neither*
CA: *false*	GN: *neither*

2.3.2. Metamodel conformity. Unlike the bacteriological approach used to automatically generate test case models from scratch in [23], the proposed approach is initialized with test case models that were created to conform to a metamodel that may contain metamodel elements that are not compatible with the target metamodel. Because of this, there may exist test case model elements that do not conform to the target metamodel, and if so, should be removed or modified to improve the validity of the test case models by reducing the number of invalid model elements.

Calculating the metamodel conformity objective value of given test case models and target metamodel is done by summing up the number of test case model elements from all test case models that do not conform to the target metamodel. For example, say that Metamodel 1.0 includes *integer* class attribute elements with the representative values *1*, *2*, *3*, and *4*. Metamodel 2.0 includes *integer* class attribute elements with representative values of *2*, *3*, *4*, and *5*. When starting from test case models that satisfy 100% of the coverage items for Metamodel 1.0 and generating test case models to conform to Metamodel 2.0, all test case model elements that are class attributes with the value *1* are invalid because they do not conform to Metamodel 2.0. These elements must be removed or modified to improve test case model validity.

2.3.3. Number of refactorings. While automatically generating test case models in an attempt to maximize metamodel coverage has been previously explored and improving metamodel conformity of test case models by itself can be accomplished trivially by removing or modifying nonconforming test case model elements, performing these tasks by finding a minimal number of refactorings to apply to existing test case models has not yet been explored to our knowledge and highlights the main contribution of this paper. By minimizing the number of refactorings required to update existing test case models to a new target metamodel, the task of updating expected test case model transformation output is simplified. The challenge of finding a minimal set of refactorings to apply to test case models to maximize metamodel coverage and minimize the number of nonconforming test case model elements, stems from the fact that there are a multitude of different refactoring sequences that can be applied to achieve the same resulting test case models. Calculating the number of refactorings is done by summing up the number of refactorings in the lists of refactorings.

2.4. Search-based Approach.

2.4.1. Simulated annealing. Simulated annealing (SA) is a local search heuristic inspired by the concept of annealing in metallurgy where metal is heated, raising its energy and relieving it of defects due to its ability to move around more easily. As its temperature drops, the metal's energy drops and eventually it settles in a more stable state and becomes rigid. This technique is replicated in SA by initializing a temperature variable with a "high temperature" value and slowly decreasing the temperature for a set number of iterations by multiplying it by a value α every iteration, where $0 < \alpha < 1$. During each iteration, a mutation operator is applied to a copy of the resulting solution from the previous iteration. If the mutated solution has the same or better fitness than the previous one, it is kept and used for the next iteration. If the mutated solution has a worse fitness, a probability of keeping the mutated solution and using it in the next iteration is calculated using an acceptance probability function which takes as input the difference in fitness of the two solutions as well as the current temperature value and outputs the acceptance probability such that smaller differences in solution fitness and higher temperature values will yield higher acceptance probabilities. In effect, this means that for each passing iteration, the probability of keeping a mutated solution with worse fitness decreases, resulting in a search policy that, in general, transitions from an explorative policy to an exploitative policy. The initial lenience towards accepting solutions with worse fitness values is what allows simulated annealing to escape local minima/maxima.

2.4.2. Multi-objective simulated annealing. Traditional SA is not suitable for the automatic test case model generation as described previously because a solution's fitness consists of three separate objective functions and SA cannot compare solutions based on multiple criteria. Furthermore, even if SA had the ability to determine relative solution fitness, there would still be the problem of quantifying the fitness disparity between solutions as a scalar value for use in the acceptance

probability function. Multi-objective simulated annealing (MOSA) overcomes these problems. When comparing the relative fitness of solutions, MOSA utilizes the idea of Pareto optimality using dominance as a basis for comparison. Solution A is said to dominate solution B if: (1) every objective value for solution A is the same or better than the corresponding objective value for solution B, and (2) solution A has at least one objective value that is strictly better than the corresponding objective value of solution B. If solution A does not dominate solution B and solution B does not dominate solution A, then these solutions are said to belong to the same non-dominating front. In MOSA, the mutated solution will be kept and used for the next iteration if it dominates or is in the same non-dominating front as the solution from the previous iteration. To determine the probability that the mutated solution dominated by the solution from the previous iteration will be kept and used for the next iteration of MOSA, there are a number of possible acceptance probability functions that can be utilized. Since previous work has noted that the average cost criteria, shown in Equation 1, yields good performance [24], we have utilized it. The average cost criteria simply takes the average of the differences of each objective value between two solutions, i and j, over all objectives D, as shown in Equation 1. The final acceptance probability function used in MOSA is shown in Equation 2.

$$c(i,j) = \frac{\sum_{k=1}^{|D|}(c_k(j) - c_k(i))}{|D|} \tag{1}$$

$$AcceptProb(i,j,temp) = e^{\frac{-abs(c(i,j))}{temp}} \tag{2}$$

2.4.3. MOSA adaptation for generating test case models. When using the number of refactorings fitness criterion along with mutations that add, modify, or remove refactorings in MOSA, a slight modification of the definition of dominance

is required in order to obtain quality results. The problem with using the traditional definition of dominance in this case is that "remove refactoring" mutations will always generate a solution that is at least in the same non-dominated front as the non-mutated solution if it does not dominate the non-mutated solution because it utilizes less refactorings, thus making it strictly better in at least one objective. In MOSA, this means that the non-mutated solution will always be discarded in favor of the mutated solution that it will use in the following iteration. The problem with this is that the probability of an add refactoring or modify refactoring mutation yielding a mutated solution that is in the same non-dominated front or better is much less than that of a mutation removing a refactoring (100%). This is because the only way an add or modify refactoring mutation could at least be in the same non-dominated front is if it satisfied a previously unsatisfied metamodel coverage item, removed an invalid model element, or modified an invalid model element to make it valid. As a result, solutions tend to gravitate towards solutions with less refactorings that eventually results in solutions with the least possible number of refactorings, one refactoring per each test case model. This was found to be the case in experiments executed with the traditional dominance implementation.

The problem is alleviated by modifying how dominance is determined in MOSA such that a mutated solution with less refactorings and less metamodel coverage or more invalid model elements than the non-mutated solution is considered to be dominated by the non-mutated solution. In other words, MOSA will only transition from the non-mutated solution from the previous iteration to the new mutated solution (using the "remove refactoring" mutation) with 100% probability if the mutated solution dominates the non-mutated solution. If the mutated solution has less refactorings but also less metamodel coverage or more invalid model elements, then it will only be accepted and used for the next iteration given the probability calculated by the acceptance probability function.

The second problem to overcome is how to use the metamodel coverage, number of invalid model elements, and number of refactoring values in the acceptance probability function in a meaningful way. As they are, these three values take on values in different scales: metamodel coverage takes on values between 0% and 100% (0.0 and 1.0), number of invalid model elements takes on values between 0 and the initial number of invalid model elements before MOSA begins, and the number of refactorings takes on the value of any nonnegative integer. In order to make the average of differences between fitness criteria values meaningful, normalization is performed. Metamodel coverage does not require any normalization as its values already lie between 0.0 and 1.0 and thus all differences between metamodel coverage values will as well. The only operation necessary is to take the absolute value of the difference to ensure it is positive as shown in Equation 3. To normalize the difference between numbers of invalid model elements, simply take the absolute value of the difference between the number of invalid model elements values and divide by the number of invalid model elements from the initial test case models as shown in Equation 4.

$$CovDiff = abs(Cov(i) - Cov(j)) \tag{3}$$

$$InvDiff = \frac{abs(Inv(i) - Inv(j))}{Inv_0} \tag{4}$$

To normalize the difference in number of refactorings, the maximum number of refactorings should be used as a divisor. Since there is theoretically no upper bound to the possible number of refactorings that the lists of refactorings could have, a reasonable estimate is required. For this estimate, the sum of the initial number of unsatisfied coverage items and the number of invalid model elements of the starting test case models is used because it assumes that each coverage item and invalid model element will take one refactoring to satisfy and remove, respectively. As shown in

Equation 5, the normalization of the difference in number of refactorings is calculated by taking the absolute value of the difference in number of refactorings divided by the sum of the initial number of unsatisfied coverage items and the number of invalid model elements of the starting test case models.

$$NumRefDiff = \frac{abs(NumRef(i) - NumRef(j))}{UnsatCovItems_0 + Inv_0} \tag{5}$$

2.5. Implementation. Before using MOSA to generate the lists of refactorings, a maximum model size must be declared to ensure a balance between the size of the test cases and the number of test cases is maintained. As explained in [23], smaller test cases allow for easier understanding and diagnosis when an error arises while the number of test cases should be reasonable in order to maintain an acceptable execution time and amount of effort for defining an oracle function.

After the maximum model size is declared, the automatic test case model generation begins. The algorithm iterates through all test case models once. For each test case model, its corresponding list of refactorings is initialized with one randomly-generated refactoring before the adapted MOSA algorithm is executed. After the algorithm has iterated over every test case model, the final lists of refactorings for each test case model are output along with the resulting test case models yielded from the application of the refactorings. The pseudocode for this algorithm is shown in Algorithm 1. It is important to note that although search is done for refactorings at the test case model level, the objective functions are executed on the overall running solution of the entire set of updated test case models at any given iteration. This means that, for example, if the space of refactoring lists for a particular test case model is being searched and a mutation is performed that covers a new coverage item for that test case model, but a list of refactorings for another test case model from a previous iteration already covered that particular coverage item, then there

is no increase in the metamodel coverage objective function. The value yielded from the metamodel coverage objective function will only increase if a coverage item is covered that has not already been covered by any other test case model with their refactorings in the overall solution.

Algorithm 1 Pseudocode for adapted MOSA for generating test case models

```
function MOSA(testCaseModels, maxModelSize, initialTemperature, α)
    ListOfRefactorings.setMaxModelSize(maxModelSize)
    solution ← list()
    for testCaseModel in testCaseModels do
        refactorings ← ListOfRefactorings(testCaseModel)
        temp ← initialTemperature
        for iteration = 1 → maxIterations do
            newRefactorings ← copy(refactorings)
            newRefactorings.mutate()
            if newRefactorings.dominates(refactorings) then
                refactorings ← newRefactorings
            else if u[0.0,1.0] < AcptProb(refactorings,newRefactorings,temp) then
                refactorings ← newRefactorings
            temp ← temp × α
        solution.push(refactorings)
    return listsOfRefactorings

function ListOfRefactorings::ListOfRefactorings(testCaseModel)
    this.testCaseModel ← testCaseModel
    this.refactorings ← list()
    this.refactorings.push(Refactoring(testCaseModel))

function ListOfRefactorings::dominates(otherList)
    if Coverage(this) > Coverage(otherList) then return True
    if NumInvElems(this) < NumInvElems(otherList) then return True
    if NumRefs(this) < NumRefs(otherList) then
        if Coverage(this) == Coverage(otherList) then
            if NumInvElems(this) == NumInvElems(otherList) then return True
    return False
```

3. EXPERIMENTATION

3.1. Experimental Setting. To test the effectiveness of the proposed approach, experiments were carried out to evolve test case models for the UML 2.0 metamodel. In the implementation used, the UML 2.0 metamodel generated 857 coverage items that needed to be satisfied in order to obtain 100% metamodel coverage. To discover if initializing the test case models with those of a previous metamodel version was beneficial, experiments were done starting from a set of test case models that conform to UML 1.4 as well as a set of "blank" test case models. The UML 1.4 test case models consist of between 17 and 23 model elements that initially satisfy 46.58% of the UML 2.0 metamodel coverage items and have 60 invalid model elements, while the blank test case models each consist of only five class model elements and satisfy 0% of the UML 2.0 metamodel coverage items and have no invalid model elements. Both sets are comprised of 20 test case models each.

To justify the multi-objective approach proposed in this paper, the same experiments were carried out using an SA approach utilizing only metamodel coverage like in previous works [23]. All experiments were run 30 times in order to establish statistical significance. For each of the 20 test case models, 10,000 iterations of SA were performed with a starting temperature of 0.0003 and an alpha value of 0.99965.

When randomly generating a mutation, each type of mutation had the same probability of being generated; there was a one-third chance each of adding a refactoring, modifying a refactoring, or removing a refactoring. If the add refactoring mutation was chosen, then there was equal chance of each type of refactoring shown in Table 4 being chosen. If the modify refactoring mutation was chosen, then there was equal chance of any refactoring in the list of refactorings being chosen for modification. For the refactoring chosen for modification, there was equal chance of the following being chosen: (1) modify a refactoring parameter, (2) replace the chosen

Table 4: Refactorings used in experiments

Add Field	Add Association	Move Field	Push Down Field
Add Method	Add Generalization	Move Method	Push Down Method
Add Class	Remove Method	Extract Class	Pull Up Field
Remove Field	Remove Association	Extract Subclass	Pull Up Method
Remove Class	Remove Generalization	Extract Superclass	Collapse Hierarchy
Change Bi- to Uni-Directional Association		Change Uni- to Bi-Directional Association	

refactoring with a randomly-generated refactoring of the same refactoring type, or (3) replace the chosen refactoring with a randomly-generated refactoring. If the remove refactoring mutation was chosen, each refactoring in the list of refactorings had equal chance of being removed.

3.2. Results. The complete results from all four experiment configurations can be found in Table 5. The SA approaches outperformed the corresponding MOSA approaches in the metamodel coverage objective as shown in Figure 3 while, however, using a far greater number of refactorings as shown in Figure 4. Figure 5 shows that the MOSA experiment that started with the UML 1.4 test case models removed all 60 test case model elements every run while the corresponding SA experiment removed less than half of the invalid test case model elements on average. All differences in results were determined to be statistically significant employing a two-tailed t-test with $\alpha = 0.05$.

Table 5: Empirical results with standard deviations in parenthesis

	Blank		**Previous Models**	
	SA	**MOSA**	**SA**	**MOSA**
% Coverage	83.82 (0.05)	63.36 (0.04)	96.20 (<0.01)	91.70 (0.01)
Invalid	-	-	35.47 (4.03)	0.00 (0.00)
Num. Ref.	1185.87 (176.69)	315.17 (18.08)	726.87 (34.15)	348.90 (13.60)

3.3. Discussion. With respect to the metamodel coverage objective, it is intuitive that the SA approaches would outperform the MOSA approaches, albeit by a relatively small margin when starting from existing test case models, because

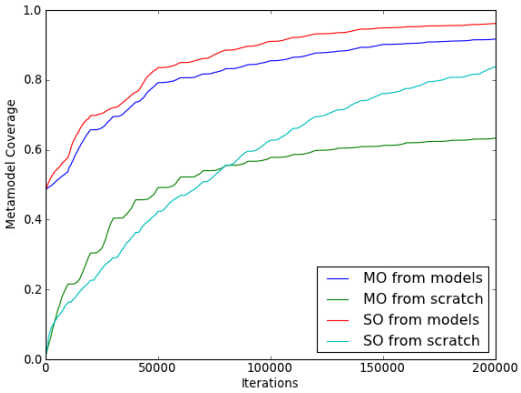

Figure 3: Metamodel coverage versus iterations

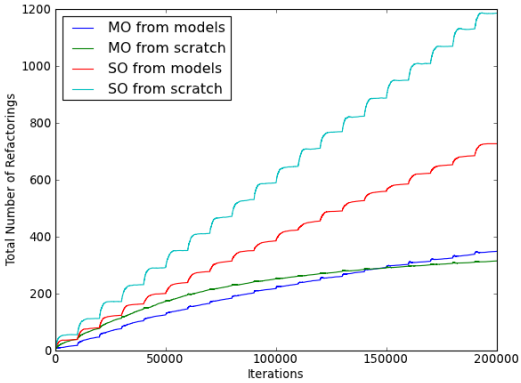

Figure 4: Refactorings versus iterations

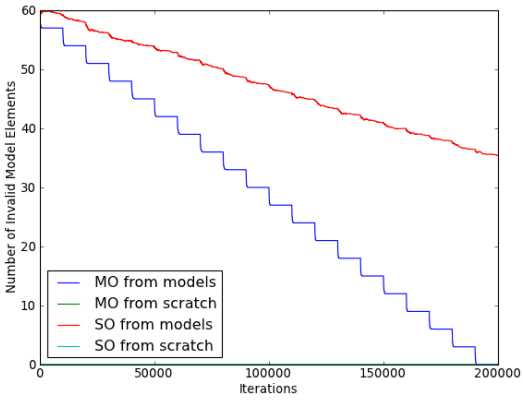

Figure 5: Invalid model elements versus iterations

the MOSA approaches must balance conflicting objectives while the SA approaches do not. As a result, the lists of refactorings yielded from the MOSA approaches are more effective in terms of metamodel coverage per refactoring than the ones yielded from SA. Combined with the fact that the total number of refactorings yielded by the MOSA approaches are drastically less than those yielded by the SA approaches, this means that the effort required to implement the changes to expected output is less and overall more effective using the MOSA approach.

Furthermore, the results show that the approaches that start with existing test case models of a previous metamodel version outperform the same approaches

that start with blank models. This also helps reduce the effort required to update the expected test case output because portions of the expected output for the existing test cases will not need to be modified.

4. RELATED WORK

In this section, contributions related to model-level test case generation and search-based testing approaches are presented. Fleurey et al. [23, 26] and Steel et al. [27] discuss the reasons why testing model transformations is distinct from testing traditional implementations: the input data are models that are complex in comparison to simple data types. Both papers describe how to generate test data in MDE by adapting existing techniques, including functional criteria [21] and bacteriologic approaches [22]. Lin et al. [28] propose a testing framework for model transformation built on their modeling tools and transformation engine that offers a support tool for test case construction, test execution, and test comparison; however, the test case models are manually developed in this work.

Some other approaches are specific to test case generation for graph transformation mechanisms. Küster [29] addresses the problem of model transformation validation in a way that is very specific to graph transformation by focusing on the verification of transformation rules with respect to termination and confluence. This approach aims to ensure that a graph transformation will always produce a unique result. Küster's work is concerned with the verification of transformation properties rather than the validation (testing) of their correctness. Darabos et al. [30] investigate the testing of graph transformations by considering graph transformation rules as the transformation specification and propose to generate test data from this specification. Darabos et al. propose several faulty models that can occur when performing

pattern matching as well as a test case generation technique that targets those particular faults. Compared to the multiobjective search-based approach proposed in this paper, Darabos work is specific to graph-based transformation testing. Mottu et al. [20] describe six different oracle functions to evaluate the correctness of an output model. In [31], the authors suggest manually determining the expected outcome of the transformation and comparing it with the actual outcome of the transformation using a simple graph-comparison algorithm.

The multi-objective search-based approach proposed in this paper is inspired by contributions in the domain of Search-Based Software Engineering (SBSE) [32]. SBSE uses search-based approaches to solve optimization problems in software engineering, and once a software engineering task is framed as a search problem, many search algorithms can be applied to solve that problem. These search-based approaches are also used to solve problems in software testing [31, 33, 34]. The general idea behind the proposed approach is that possible test case model refactorings define a search space and multiple conflicting test case model criteria are integrated into multiple objective functions. These components guide the search approach in an attempt to find an optimal set of test case model refactorings that yields a set of adequate updated test case models.

To conclude, although the problem of generating test cases at the code level is well-studied, there are few works that generate test cases at the model level to test transformation mechanisms. To our knowledge, there is currently no other work that utilizes existing test case models of a previous metamodel version to generate test case models for an updated metamodel version. Furthermore, this is the first adaptation of heuristic search algorithms to take into consideration multiple objectives when generating artificial source models (test cases) similar to the data that will be transformed.

5. CONCLUSION AND FUTURE WORK

Empirical results show that MOSA can automatically generate quality test case models from existing test case models in response to metamodel changes. The new test case models are generated in such a way that the effort required to update the expected test case model transformation output is reasonable. While SA is able to achieve better overall metamodel coverage, the effort required to update the expected test case model transformation results is not reasonable. Furthermore, the MOSA approach is able to reliably remove test case model elements that become invalid due to metamodel changes.

To generalize our proposed approach and ensure its robustness, we plan to extend our validation to other metamodels such as Petri nets and relational schema. Furthermore, comparative studies between different multiobjective metaheuristic algorithms will be performed.

SECTION

2. CONCLUSIONS

In this thesis, two novel approaches are presented that show how CI can be used to improve the scalability other CI algorithms through hybridization as well as specific application domains, namely, model transformation testing in software engineering. As presented in the first paper, the novel hybridization of XCS and cluster analysis provides a means of improving the scalability of XCS functionality, making its utilization with larger and more complex problems more feasible. The novel application of MOSA to test case model generation in the second paper allows for the efficient generation of new test case models in response to metamodel changes that minimizes the effort required to redefine expected output, thus making it more practical and scalable in real-world problems.

As the number, size, and complexity of real-world problems continually grow, improvements such as these are critical to the success of CI techniques because they increase the accessibility of CI techniques for utilization on a wider array of problems. As ongoing research continues to improve the overall accessibility of CI techniques to problems of all types, the main consideration when solving a complex problem shifts from choosing the best CI technique that happens to support a particular problem configuration to choosing the best CI technique based solely on the suitability of its underlying problem solving mechanism. As this shift becomes more prevalent, further research into methods of choosing the most suitable CI technique for a given problem should reveal insights that can be used to predict the relative effectiveness

of CI techniques on both specific problems as well as general problem classes. When enough insight is revealed that yield highly accurate CI effectiveness predictions, problem solving can be available to a wider audience of non-CI experts through automated processes. As a result, experts in non-CI domains will be able to take advantage of CI techniques' ability to solve complex problems and apply it to their domain of expertise without requiring the assistance of a CI expert.

BIBLIOGRAPHY

[1] Matthew Gershoff and Sonia Schulenburg. Collective Behavior-based Hierarchical XCS. In *Proceedings of the 2007 GECCO Conference Companion on Genetic and Evolutionary Computation*, GECCO '07, pages 2695–2700, New York, NY, USA, 2007. ACM.

[2] Urban Richter, Holger Prothmann, and Hartmut Schmeck. Improving XCS Performance by Distribution. In *Proceedings of the 7th International Conference on Simulated Evolution and Learning*, SEAL '08, pages 111–120, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Mani Abedini and Michael Kirley. CoXCS: A Coevolutionary Learning Classifier Based on Feature Space Partitioning. In *Proceedings of the 22nd Australasian Joint Conference on Advances in Artificial Intelligence*, AI '09, pages 360–369, Berlin, Heidelberg, 2009. Springer-Verlag.

[4] M. Abedini and M. Kirley. A Multiple Population XCS: Evolving Condition-Action Rules based on Feature Space Partitions. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1 –8, July 2010.

[5] L. Bull, M. Studley, A. Bagnall, and I. Whittley. Learning Classifier System Ensembles with Rule-Sharing. *Evolutionary Computation, IEEE Transactions on*, 11(4):496–502, 2007.

[6] Fani A. Tzima, Pericles A. Mitkas, and John B. Theocharis. Clustering-based Initialization of Learning Classifier Systems: Effects on Model Performance, Readability and Induction Time. *Soft Comput.*, 16(7):1267–1286, July 2012.

[7] Zbigniew Maciej Skolicki. *An Analysis of Island Models in Evolutionary Computation*. PhD thesis, George Mason University, Fairfax, Virginia, U.S.A., 2007.

[8] StewartW. Wilson. Compact Rulesets from XCSI. In PierLuca Lanzi, Wolfgang Stolzmann, and StewartW. Wilson, editors, *Advances in Learning Classifier Systems*, volume 2321 of *Lecture Notes in Computer Science*, pages 197–208. Springer Berlin Heidelberg, 2002.

[9] Phillip William Dixon, David Wolfe Corne, and Martin John Oates. A Ruleset Reduction Algorithm for the XCS Learning Classifier System. In PierLuca Lanzi, Wolfgang Stolzmann, and StewartW. Wilson, editors, *Learning Classifier Systems*, volume 2661 of *Lecture Notes in Computer Science*, pages 20–29. Springer Berlin Heidelberg, 2003.

[10] Muhammad Iqbal, WillN. Browne, and Mengjie Zhang. Evolving Optimum Populations with XCS Classifier Systems. *Soft Computing*, 17(3):503–518, 2013.

[11] E. W. Forgy. Cluster Analysis of Multivariate Data: Efficiency vs Interpretability of Classifications. *Biometrics*, 21:768–769, 1965.

[12] J. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In *Proc. Fifth Berkeley Symp. on Math. Statist. and Prob.*, volume 1, pages 281–297. Univ. of Calif. Press, 1967.

[13] T. Calinski and J. Harabasz. A Dendrite Method for Cluster Analysis. *Communications in Statistics - Theory and Methods*, 3(1):1–27, 1974.

[14] Rui Xu and Don Wunsch. *Clustering*. Wiley-IEEE Press, 2009.

[15] Glenn W Milligan and Martha C Cooper. An Examination of Procedures for Determining the Number of Clusters in a Data Set. *Psychometrika*, 50(2):159–179, 1985.

[16] Jacob Goldberger, Sam Roweis, Geoff Hinton, and Ruslan Salakhutdinov. Neighbourhood Components Analysis. In *Advances in Neural Information Processing Systems 17*, pages 513–520. MIT Press, 2004.

[17] Stewart W. Wilson. Get Real! XCS with Continuous-Valued Inputs. In *Learning Classifier Systems, From Foundations to Applications, LNAI-1813*, pages 209–219. Springer-Verlag, 2000.

[18] A. Frank and A. Asuncion. UCI Machine Learning Repository, 2010.

[19] Martin Butz and Stewart W. Wilson. An Algorithmic Description of XCS. In *Revised Papers from the Third International Workshop on Advances in Learning Classifier Systems*, IWLCS '00, pages 253–272, London, UK, UK, 2001. Springer-Verlag.

[20] J.M. Mottu, B. Baudry, and Y. Le Traon. Model Transformation Testing: Oracle Issue. In *IEEE International Conference on Software Testing Verification and Validation Workshop, 2008. ICSTW '08*, pages 105–112, 2008.

[21] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. le Traon. Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In *ISSRE '06, 17th International Symposium on Software Reliability Engineering*, pages 85–94, 2006.

[22] Benoit Baudry, Franck Fleurey, Jean-Marc Jezequel, and Yves Le Traon. Automatic Test Cases Optimization Using a Bacteriological Adaptation Model: Application to .NET Components. In *Proceedings of ASE'02 (Automated Software Engineering), Edimburgh*, 2002.

[23] F. Fleurey, J. Steel, and B. Baudry. Validation in Model-Driven Engineering: Testing Model Transformations. In *Proceedings of First International Workshop on Model, Design and Validation*, pages 29–40, 2004.

[24] Dongkyung Nam and Cheol Hoon Park. Multiobjective Simulated Annealing: A Comparative Study to Evolutionary Algorithms. *International Journal of Fuzzy Systems*, 2(2):87–97, 2000.

[25] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards Scenario-Based Testing of UML Diagrams. In Achim D. Brucker and Jacques Julliand, editors, *Tests and Proofs*, volume 7305 of *Lecture Notes in Computer Science*, pages 149–155. Springer Berlin Heidelberg, 2012.

[26] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and YvesLe Traon. Qualifying input test data for model transformations. *Software & Systems Modeling*, 8(2):185–203, 2009.

[27] J. Steel and M. Lawley. Model-based Test Driven Development of the Tefkat Model-Transformation Engine. In *15th International Symposium on Software Reliability Engineering. ISSRE 2004.*, pages 151–160, 2004.

[28] Yuehua Lin, Jing Zhang, and Jeff Gray. A Testing Framework for Model Transformations. In *Research and Practice in Software Engineering - Model-Driven Software Development. 2005*, pages 219–236. Springer, 2005.

[29] Jochen M. Küster and Mohamed Abd-El-Razik. Validation of Model Transformations – First Experiences using a White Box Approach. In *Proceedings of MODEVA 06 (Model Design and Validation Workshop Associated to MODELS 06)*, pages 193–204. Springer, 2006.

[30] Andrea Darabos, András Pataricza, and Dániel Varró. Towards Testing the Implementation of Graph Transformations. In *In Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques*, pages 69–80. Elsevier, 2006.

[31] Phil McMinn. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[32] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search-based Software Engineering: Trends, Techniques and Applications. *ACM Comput. Surv.*, 45(1):11:1–11:61, December 2012.

[33] André Baresel, David Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: a testability transformation approach. In *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, pages 108–118, New York, NY, USA, 2004. ACM.

[34] André Baresel, Harmen Sthamer, and Michael Schmidt. Fitness Function Design To Improve Evolutionary Structural Testing. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '02, pages 1329–1336, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

VITA

Jeffery Scott Shelburg grew up in Saint Charles, Missouri. He achieved the rank of Eagle Scout in the Boy Scouts of America and graduated Valedictorian from Saint Charles West High School. From Fall of 2008 to Spring of 2011, he attended Missouri University of Science and Technology to earn a Bachelor of Science degree in Computer Science with the help of Bright Flight, AT&T Foundations, and Missouri S&T Curator's scholarships. From Spring 2009 to Fall of 2012, Jeffery served as an officer of Missouri S&T's ACM SIG-Security organization as University Relations officer and then Co-Chair. In the Summer of 2010, Jeffery worked for Sandia National Laboratories as a technical intern in the Center for Cyber Defenders. He was then accepted into Sandia's Critical Skills Master's Program that lasted from the Summer of 2011 to Spring of 2013 during which he worked at Sandia in the summers. With funding from Sandia's Critical Skills Master's Program, Jeffery earned his Master of Science degree in Computer Science from Missouri University of Science and Technology in Spring of 2013 and performed the research upon which the two papers in this thesis were based.