Maximizing Refactoring Coverage in an Automated Maintenance Approach using Multi-Objective Optimization

Michael Mohan, Des Greer and Paul McMullan
School of Electronics, Electrical Engineering and Computer Science
Queen's University Belfast
United Kingdom
{mmohan03|des,greer|p.p.mcmullan}@qub.ac.uk

***Abstract*—This paper describes a multi-objective genetic algorithm used to automate software refactoring. The approach is validated using a set of open source Java programs with a purpose built tool, MultiRefactor. The tool uses a metric function to measure quality in a software system and tests a second objective to measure the amount of code coverage of the applied refactorings by analyzing the code elements they have been applied to. The multi-objective setup will refactor the input program to improve its quality using the quality objective, while also maximizing the code coverage of the refactorings applied to the software. An experiment has been constructed to measure the multi-objective approach against the alternative mono-objective approach that does not use an objective to measure refactoring coverage. The two approaches are tested on six different open source Java programs. The multi-objective approach is found to give significantly better refactoring coverage scores across all inputs in a similar time, while also generating improvements in the quality scores.**

Keywords—search based software engineering, maintenance, automated refactoring, refactoring tools, software quality, multi-objective optimization, genetic algorithms

I. Introduction

An increasing proportion of Search Based Software Maintenance (SBSM) research is making use of multi-objective optimization techniques. Many multi-objective search algorithms are built using genetic algorithms (GAs), due to their ability to generate multiple possible solutions. Instead of focusing on only one property, the multi-objective algorithm is concerned with a number of different objectives. This is handled through a fitness calculation and sorting of the solutions after they have been modified or added to. The main approach used to organize solutions in a multi-objective approach is Pareto dominance [1]. Pareto dominance organizes the possible solutions into different non-domination levels and further discerns between them by finding the objective distances between them in Euclidean space. If there are more than three objectives used, Pareto dominance becomes inefficient [2]–[4] as an increasingly larger fraction of the population becomes non-dominated and the solutions generated will be less diverse. Many-objective algorithms concerning more than three objectives have overcome this issue with various techniques like objective reduction [5], alternative preference ordering relations [6], decomposition [7], use of a predefined multiple targeted search [8] and the incorporation of a decision maker's preferences [9].

In this paper, a multi-objective approach is created to improve software that combines a quality objective with one that measures the extent of coverage that a refactoring solution can give among the elements of the solution. Coverage is important since a developer may not want the solution to focus on only a few parts of the code or get stuck on certain areas. Ensuring good coverage also avoids the possibility of a solution focusing more on specific areas of a class performing redundant refactorings. As the MultiRefactor tool has many complimentary refactorings available (such as Make Class Abstract/Make Class Concrete), it is possible that a solution will have a number of refactorings that are applied to the same elements and that reverse the effects of the refactorings that come before it, causing the effect to be meaningless. Having an objective measurement to increase refactoring coverage reduces the likelihood of redundant refactorings.

A refactoring coverage objective has been constructed within the tool to inspect the refactoring solutions generated as part of the genetic search, and rank their fitness by analyzing the refactorings applied and calculating a coverage score. The coverage score will be determined by two factors. The more elements inspected within a refactoring solution, the better the score will be. These elements include classes, methods and fields/variables. For each refactoring, a single element will be chosen to correspond to it, where class level refactorings will choose the relevant class, and likewise, method and field level refactorings will choose the relevant method or field. The number of distinct elements corresponding to the refactorings in a solution can be calculated this way and therefore it can be determined which solution looks at more. The second factor inspected will be the number of times each element is refactored. The smaller the average number of refactorings for each element in a solution, the better the score will be. This way, the score will minimize the effect of solutions with a larger number of refactorings and encourage the solution to focus less on a specific element or group of elements. This will also minimize the occurrence of the redundant refactorings discussed above and allow each refactoring to have meaning in the solution by dispersing the refactorings across the different elements. To test the effectiveness of the refactoring coverage objective, an experiment has been constructed to test a GA that uses it against one that does not. In order to judge the outcome of the experiment, the following research questions have been derived:

RQ1: Does a multi-objective solution using a refactoring coverage objective and a quality objective give an improvement in quality?

RQ2: Does a multi-objective solution using a refactoring coverage objective and a quality objective improve the

978-1-7281-2270-0/19/$31.00 ©2019 IEEE
DOI 10.1109/IWoR.2019.00014

coverage of refactorings better than a solution that does not use the refactoring coverage objective.

In order to address the research questions, the experiment will run a set of tasks to compare a default mono-objective set up to refactor a solution towards quality with a multi-objective approach that uses a quality objective and the newly proposed refactoring coverage objective. The following null hypotheses have been constructed to assess success in the experiment.

$H1_0$: The multi-objective solution does not give an improvement in the quality objective value.

$H2_0$: The multi-objective solution does not give significantly higher refactoring coverage values than the corresponding mono-objective solution.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 describes the MultiRefactor tool used to conduct the experimentation along with the searches, refactorings and metrics available in it. Section 4 explains the set up of the experiment used to test the refactoring coverage objective, as well as the outcome of previous experimentation done to derive the quality objective and the GA parameters used. Section 5 analyzes the results of the experiment, looking at the objective values and the times taken to run the tasks. Section 6 outlines the threats to validity and concludes the paper.

II. Related Work

A number of recent studies in SBSM have explored the use of multi-objective techniques. White et al. [10] used a multi-objective approach to find a tradeoff between the functionality of a pseudorandom number generator and power consumption required. De Souza et al. [11] investigated the human competitiveness of SBSE techniques in 4 areas of software engineering using mono-objective and multi-objective GAs. Ouni et al. [12] created an approach to measure semantics preservation in a software program when searching for refactoring options to improve the structure, by using the NSGA-II search. Ouni et al. [13], [3], [14] then explored the potential of using development refactoring history to aid in refactoring a software project using NSGA-II. In another study [15], Ouni et al. combined this approach in a four objective configuration that also aimed to minimize the number of code changes necessary to fix defects. Ouni et al. [16] also expanded upon the code smells correction approach of Kessentini et al. [17] by replacing the GA used with NSGA-II. Wang et al. also expanded on the approach of Kessentini et al. by combining the detection and removal of software defects with an estimation of the number of future code smells generated in the software by the refactorings. Ouni et al. [18] investigated the use of a chemical reaction optimization algorithm to explore the benefits of this approach for SBSM. They compared the algorithm against three other search algorithms including a GA. Mkaouer et al. [19], [20] experimented with combining quality measurements with robustness using NSGA-II. Mkaouer et al. [2], [21], [22] also used NSGA-III to experiment with automated maintenance.

III. MultiRefactor

The MultiRefactor approach[1], in common with those of Moghadam and O' Cinnéide [23] and Trifu et al. [24], uses the RECODER framework [2] to modify source code in Java programs. RECODER extracts a model of the code (represented using an Abstract Syntax Tree) that can be used to analyze and modify the code before the changes are applied and written to file. MultiRefactor combines mono-objective search optimization algorithms like simulated annealing with a multi-objective genetic algorithm (MOGA) to make available various different approaches to automated software maintenance in Java programs. It takes Java source code as input and will output the modified source code to a specified folder. The input must be fully compilable and must be accompanied by any necessary library files as compressed jar files. The numerous searches available in the tool have various input configurations that can affect the execution of the search. The refactorings and metrics used can also be specified. As such, the tool can be configured in a number of different ways to specify the particular set up you want to use in relation to the selection of metrics and refactorings, the search used and the settings chosen for that particular search). If desired, multiple tasks can be set to run one after the other.

A previous study [25] used the A-CMA [26] tool to experiment with different metric functions but that work was not extended to produce output source code (likewise, TrueRefactor [27] only modifies UML and Ouni et al.'s [16] approach only generates proposed lists of refactorings). MultiRefactor was developed in order to be a fully-automated search-based refactoring tool that produces compilable, usable code. As well as the Java code artifacts, the tool will produce an output file that gives information on the execution of the task including data about the parameters of the search executed, the metric values at the beginning and end of the search, and details about each refactoring applied. The metric configurations can be modified to include different weights and the direction of improvement of the metrics can be changed depending on the desired outcome. These configurations can be read in a number of ways including as text files or xml files. There are a few ways the metric functions can be calculated. An overall metric value can be found using a weighted metric sum or Pareto dominance can be used to compare individual metrics within the functions.

MultiRefactor contains seven different search options for automated maintenance, with three distinct metaheuristic search techniques available. For each search type there is a selection of configurable properties to determine how the search will run. The types of search available are detailed in Table 1. The MOGA used in the experimentation is largely identical to the basic GA, and contains the same configuration options. The representation used in MultiRefactor is based on the implementation used by Seng et al. [28] and further adapted by O' Keefe and O' Cinnéide [29]. The search algorithm stores model information to represent multiple different genomes in a population, avoiding the expensive memory costs needed to store multiple different models. The

[1] https://github.com/mmohan01/MultiRefactor

[2] http://sourceforge.net/projects/recoder

initial population is constructed by applying a selection of random refactorings to the initial model to create a single genome, and repeating for the required number of genomes.

TABLE 1 – AVAILABLE SEARCH TECHNIQUES IN MULTIREFACTOR

Search Type	Search Variation
Random Search	N/A
Hill Climbing Search	First Ascent
	Steepest Ascent
Simulated Annealing	N/A
Genetic Algorithm	Simple Mono-Objective GA
	NSGA-II
	NSGA-III

For the crossover and mutation processes, each genome in the population is represented as the sequence of refactorings that have been applied. The crossover process uses the cut and splice technique, generating two offspring from two different parent genomes. The parent genomes are chosen using a selection operator. With the basic GA, rank selection is used, which gives the genomes with better fitness a higher chance of being chosen for crossover. A single, separate point in the refactoring sequence is chosen for each parent in order to facilitate the technique. The point is chosen at random for each genome, and for both, at least one refactoring will be present on each side of the chosen point. Each child is then generated by mixing the two sets of refactorings together. The first set of refactorings in one parent will be paired with the second set of refactorings from the other parent and vice versa. The refactorings will be applied consecutively to construct each child genome, and any inapplicable refactorings during this process will be left out although the genome will still be created using the remaining refactorings.

The mutation process will randomly choose from the new offspring and apply a single random refactoring at the end of the refactoring sequence for that genome. Crossover will be applied to the population of genomes at least once during each generation and may happen more depending on the input parameters specified. Likewise, mutation will be applied a certain number of times each generation depending on the parameters specified, or may not happen at all. In order to choose parent genomes for crossover, a rank selection operator is used. Once the mutation process is complete for a generation, the new offspring is added to the current population and the solutions are ordered according to fitness. The number of generations specified will determine when the search terminates and the preset population size will determine how many genomes are generated at initialization and how many will survive each generation. The crossover and mutation probabilities determine the likeliness of these processes being executed during the search. The refactoring range will determine the initial number of refactorings applied to the genomes during the initialization process. For each initial solution, a random number of refactorings between one and the preset refactoring range will be chosen.

The multi-objective algorithm is an adaptation of the NSGA-II [30] algorithm, differing from a basic GA mostly in how the fitness score is calculated. Due to the difference in how fitness is calculated, the selection operator used for crossover is different as well. Binary tournament selection is used, in order to avoid the need to rely on ranks. The multi-objective algorithm allows the user to choose multiple metric functions as separate objectives to guide the search. The genomes in the population will then be sorted using a non-dominated approach, allowing each objective to be considered separately. Unlike the approach used by Ouni et al. [16], the refactorings used will be checked for semantic coherence as part of the search, and will be applied automatically, eliminating the need to check and apply the refactorings manually, ensuring the process is fully automated.

The refactorings used in the tool are mostly based on Fowler's list [31], consisting of 26 field-level, method-level and class-level refactorings, as listed in Table 2. Each refactoring will initially deduce whether a program element can be refactored. It will make all the relevant semantic checks and return true or false to reflect whether the refactoring is applicable. The RECODER framework allows the tool to apply the changes to the element in the model. This may consist of a single change or, as in the case of the more complex refactorings, may include a number of individual changes to the model. Specific changes applied with the RECODER framework consist of either adding an element to a parent element, removing an element from a parent element, or replacing one element with another in the model. The refactoring itself is constructed using these specific model changes. The metrics in the tool measure the current state of a program and are used to assess whether an applied refactoring has had a positive or negative impact. Due to the multi-objective capabilities of MultiRefactor, the metrics can be measured as separate objectives to be more precise in measuring their effect on a program. A number of the metrics available in the tool are adapted from the list of metrics in the QMOOD [32] and CK/MOOSE [33] metrics suites. Table 3 lists the 23 metrics currently available in the tool.

TABLE 2 – AVAILABLE REFACTORINGS IN THE MULTIREFACTOR TOOL

Field Level	Method Level	Class Level
Increase Field Security	Increase Method Security	Make Class Final
Decrease Field Security	Decrease Method Security	Make Class Non Final
Make Field Final	Make Method Final	Make Class Abstract
Make Field Non Final	Make Method Non Final	Make Class Concrete
Make Field Static	Make Method Static	Extract Subclass
Make Field Non Static	Make Method Non Static	Collapse Hierarchy
Move Field Down	Move Method Down	Remove Class
Move Field Up	Move Method Up	Remove Interface
Remove Field	Remove Method	

TABLE 3 – AVAILABLE METRICS IN THE MULTIREFACTOR TOOL

QMOOD Based Metrics	CK Based Metrics	Others
Class Design Size	Weighted Methods Per Class	Abstractness
Number of Hierarchies	No. Children	Abstract Ratio
Avg. No. Ancestors		Static Ratio
Data Access Metric		Final Ratio
Direct Class Coupling		Constant Ratio
Cohesion Among Methods		Inner Class Ratio
Aggregation		Referenced Methods Ratio
Functional Abstraction		Visibility Ratio
No. Polymorphic Methods		Lines of Code

QMOOD Based Metrics	CK Based Metrics	Others
Class Interface Size		No. Files
Number of Methods		

In order to implement the refactoring coverage objective, extra information about the refactorings is stored in the refactoring sequence object used to represent a refactoring solution. For each solution, a hash table is used to store a list of affected elements in the solution and to attach to each a value that represents the number of times that particular element is refactored in the solution. During each refactoring, an element, considered to be most relevant to that refactoring, is chosen and the element name is stored. When it is extracted from the refactoring after it has executed and stored in the hash table for the solution, the hash table is inspected. If the element name already exists as a key in the hash table, the value corresponding to that key is incremented to represent another refactoring being applied to that element in the solution. Otherwise, the element name is added to the table and the corresponding value is set to 1. After the solution has been created, the hash table will have a list of all the elements affected and the number of times for each. This information can then be used to construct the coverage score for that solution. More information about the coverage score is given in section 4.

IV. Experimental design

In order to evaluate the effectiveness of the refactoring coverage objective, a set of tasks were set up that used the refactoring coverage objective to be compared against a set of tasks that didn't. The control group is made up of a mono-objective approach that uses a function to represent quality in the software. The corresponding tasks use the multi-objective algorithm and have two objectives. The first objective is the same function for software quality as used for the mono-objective tasks. The second objective is the refactoring coverage objective. The metrics used to construct the quality function and the configuration parameters used in the GAs are taken from previous experimentation on software quality. Each metric available in the tool was tested separately in a GA to deduce which were more successful, and the most successful were chosen for the quality function. The metrics used in the quality function are given in Table 4. No weighting is applied for any of the metrics. The configuration parameters used for the mono-objective and multi-objective tasks were derived through trial and error and are outlined in Table 5. The hardware used to run the experiment is outlined in Table 6.

TABLE 4 – METRICS USED IN THE SOFTWARE QUALITY OBJECTIVE

Metrics	Direction
Data Access Metric	+
Direct Class Coupling	-
Cohesion Among Methods	+
Aggregation	+
Functional Abstraction	+
No. Polymorphic Methods	+
Class Interface Size	+
No. Methods	-
Weighted Methods Per Class	-
Abstractness	+
Abstract Ratio	+
Static Ratio	+
Final Ratio	+
Constant Ratio	+
Inner Class Ratio	+
Referenced Methods Ratio	+
Visibility Ratio	-
Lines f Code	-

For the tasks, six different open source programs are used as inputs to ensure a variety of different domains are tested. The programs range in size from relatively small to medium sized. These programs were chosen as they have all been used in previous SBSM studies and so comparison of results is possible. Mango is a Java library, loosely inspired by the C++ standard template library. Beaver is a parser generator. Apache XML-RPC is a Java implementation of XML-RPC that uses XML to implement remote procedure calls. JHotDraw is a two-dimensional graphics framework for structured drawing editors. GanttProject is a tool for project scheduling and management. Finally, XOM is a tree based API for processing XML. The source code and necessary libraries for all of the programs are available to download in the GitHub repository for the MultiRefactor tool. Each one is run five times for the mono-objective approach and five times for the multi-objective approach, resulting in 60 tasks overall. The inputs used in the experiment as well as the number of classes and lines of code they contain are given in Table 7.

In order to find the coverage score for the mono-objective approach to compare against the multi-objective approach, the mono-objective GA has been modified to output the coverage score after the task finishes. At the end of the search, after the results have been output and the refactored population has been written to Java code files, the coverage score for the top solution in the final population is calculated. Then, before the search terminates, this score is output at the end of the results file for that solution. This way the scores don't need to be calculated manually and the coverage scores for the mono-objective solutions can be compared against their multi-objective counterpart.

TABLE 5 – GA CONFIGURATION SETTINGS

Configuration Parameter	Value
Crossover Probability	0.2
Mutation Probability	0.8
Generations	100
Refactoring Range	50
Population Size	50

TABLE 6 – HARDWARE DETAILS FOR THE EXPERIMENTATION

Operating System	Microsoft Windows 7 Enterprise Service Pack 1
System Type	64-bit
RAM	8.00GB
Processor	Intel Core i7-3770 CPU @ 3.40GHz

TABLE 7 – JAVA PROGRAMS USED IN THE EXPERIMENTATION

Name	LOC	Classes
Mango	3,470	78
Beaver 0.9.11	6,493	70
Apache XML-RPC 2.0	11,616	79
JHotDraw 5.3	27,824	241
GanttProject 1.11.1	39,527	437
XOM 1.2.1	45,136	224

For the quality function the metric changes are calculated using a normalization function. This function causes any

greater influence of an individual metric in the objective to be minimized, as the impact of a change in the metric is influenced by how far it is from its initial value. The function finds the amount that a particular metric has changed in relation to its initial value at the beginning of the task. These values can then be accumulated depending on the direction of improvement of the metric (i.e. whether an increase or a decrease denotes an improvement in that metric) and the weights given to provide an overall value for the metric function or objective. A negative change in the metric will be reflected by a decrease in the overall function/objective value. In the case that an increase in the metric denotes a negative change, the overall value will still decrease, ensuring that a larger value represents a better metric value regardless of the direction of improvement. In the case that the initial value of a metric is 0, the initial value used is changed to 0.01 in order to avoid issues with dividing by 0. This way, the normalization function can still be used on the metric and its value still starts off low. Equation 1 defines the normalization function, where Cm is the current metric value and Im is the initial metric value. Wm is the applied weighting for the metric and D is a binary constant that represents the direction of improvement of the metric. n represents the number of metrics used in the function. For the refactoring coverage objective, this normalization function is not needed. The objective score depends on the number of elements refactored in a solution and will reflect that.

$$\sum_{m=o}^{n} D.W_m\left(\frac{C_m}{I_m} - 1\right) \quad (1)$$

The tool has been updated in order to use a heuristic to choose a suitable solution out of the final population to inspect. The heuristic used is similar to the method used by Deb and Jain [8] to construct a linear hyper-plane in the NSGA-III algorithm. Firstly, the solutions in the population from the top rank are isolated and written to a separate sub folder. It is from this subset that the best solution will be chosen from when the task is finished. Among these solutions, the tool inspects the individual objective values, and for each, the best objective value across the solutions is stored. This set of objective values is the ideal point $\overline{z} = (z_1^{max}), (z_2^{max}), \ldots, (z_M^{max})$, where (z_i^{max}) represents the maximum value for an objective, and an objective i = 1, 2, ..., M. This is the best possible state that a solution in the top rank could have. After this is calculated, each objective score is compared with its corresponding ideal score. The distance of the objective score from its ideal value is found, i.e. $(z_i^{max}) - f_i(x)$, where $f_i(x)$ represents the score for a single objective. For each solution, the largest objective distance (i.e. the distance for the objective that is furthest from its ideal point) is stored, i.e. $fmax(x) = max_{i=1}^{M}[(z_i^{max}) - f_i(x)]$. At this point each solution in the top rank has a value, $fmax(x)$, to represent the furthest distance among its objectives from the ideal point. The smallest among these values, $min_{j=0}^{N-1} fmax(x)$ (where N represents the number of solutions in the top rank), signifies the solution that is closest to that ideal point, taking all of the objectives into consideration. This solution is then considered to be the most suitable solution and is marked as such when the population is written to file. On top of this, the results file for the corresponding solution is also updated to mark it as the most suitable. This is how solutions are chosen among the final population for the multi-objective tasks to compare against the top mono-objective solution.

For the refactoring coverage objective, the number of elements refactored is counted and then divided by the average number of times each element is refactored in order to get an overall score. This allows the refactoring coverage objective to take into account both the number of elements refactored, and the number of times an element is refactored. The score will prioritize solutions that have a larger amount of code coverage in their refactorings, and that have refactored as many elements as possible. Equation 2 gives the formula used to calculate the coverage score in a refactoring solution using the hash table structure. n represents the number of elements refactored in the refactoring solution, and A represents the number of times an element has been refactored in the solution.

$$n^2/(\sum_{m=o}^{n} A) \quad (2)$$

V. Results

Fig. 1 gives the average quality gain values for each input program used in the experiment with the mono-objective and multi-objective approaches. In all of the inputs, the mono-objective approach gives a better quality improvement than the multi-objective approach. For the multi-objective approach all the runs of each input were able to give an improvement for the quality objective as well as look at the refactoring coverage objective. For both approaches, the smallest improvement was given with GanttProject. The inputs with the largest improvements were different for each approach. For the mono-objective approach it was Beaver, whereas, for the multi-objective approach, it was XOM.

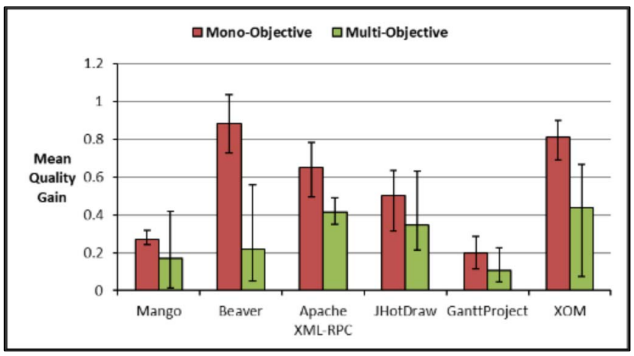

Fig. 1. Mean Quality Gain Values for Each Input

Fig. 2 shows the average coverage scores for each input with the mono-objective and multi-objective approaches. For all of the inputs, the multi-objective approach was able to yield better scores coupled with the refactoring coverage objective. The values were compared for significance using a one-tailed

Wilcoxon rank-sum test (for unpaired data sets) with a 95% confidence level ($\alpha = 5\%$). The coverage scores for the multi-objective approach were found to be significantly higher than the mono-objective approach. With the multi-objective approach, the average scores mostly increased as the input program sizes increased. This makes sense as the larger programs will contain more refactorable elements and classes in which to apply the refactorings in a solution. The notable exception to this is XOM, which had an average coverage score that is smaller than JHotDraw. This is likely due to the number of classes in the project being smaller than the GanttProject and JHotDraw class sizes. While the number of lines of code in XOM is greater, it may be that the number of code elements in the program is smaller. The scores seemed to vary slightly less with the multi-objective approach compared to the mono-objective counterparts. Again, this is understandable as the refactoring coverage objective used in the multi-objective approach to improve the program will drive the solutions towards more diverse sets of refactorings, pushing the coverage scores towards a higher peak. On the other hand, the mono-objective coverage scores are more likely to be achieved as a by-product of the other objective, leading to more fluctuating sets of scores among the tasks.

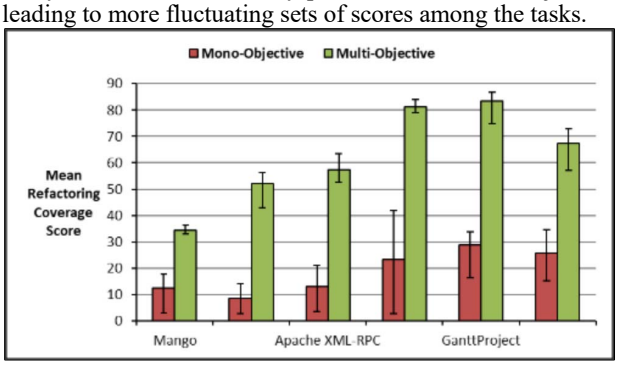

Fig. 2. Mean Refactoring Coverage Scores for Each Input

Fig. 3 gives the average execution times for each input with the mono-objective and multi-objective searches. The times for the mono-objective and multi-objective tasks seemed to mirror each other. For most input programs, the mono-objective approach was faster on average, with the exception to this being Apache XML-RPC. The Wilcoxon rank-sum test (two-tailed) was used again and the values were found to not be significantly different. Again, the times generally increased as the project sizes increased, except for XOM, where the times where smaller than both JHotDraw and GanttProject. Once again these programs, while smaller, contain more classes than XOM, which may have contributed to their increased execution times. The GanttProject program stands out as taking the longest, with the longest tasks taking over 45 minutes to run, whereas the longest tasks from the other input programs took little over 30 minutes. GanttProject has the largest number of classes, which is almost double the amount that XOM contains. Similarly, the execution times for GanttProject are around twice as large as those of XOM for the 2 approaches.

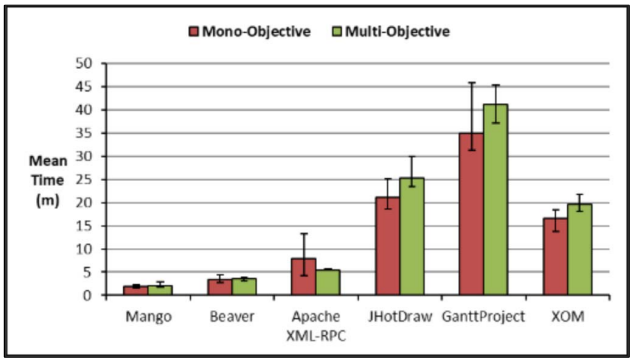

Fig. 3. Mean Times Taken for Each Input

VI. Conclusions

A. Threats to Validity

Internal validity focuses on the causal effect of the independent variables on the dependent variables. The stochastic nature of the search techniques means that each run will provide different results. This threat to validity has been addressed by running each of the tasks across 6 different open source programs and running against each program 5 times. Average values are then used to compare against each other. The choice of parameter settings used by the search techniques can also provide a threat to validity due to the option of using poor input settings. This has been addressed by using input parameters deemed to be most effective through trial and error via previous experimentation.

External validity is concerned with how well the results and conclusions can be generalized. In this study, the experiment was performed on 6 different real world open-source systems belonging to different domains and with different sizes and complexities. However, the experiment and the capabilities of the refactoring tool used are restricted to Java programs, therefore no claims can be made on generalizability to other domains or programming languages.

Construct validity refers to how well the concepts and measurements are related to the experimental design. The validity of the experiment is limited by the metrics used, as they are experimental approximations of software quality, as well as the refactoring coverage objective used to measure the number of elements refactored. The cost measures used in the experiment can also indicate a threat to validity. Part of the effectiveness of the two search approaches was measured using execution time in order to measure and compare cost.

Conclusion validity looks at the degree to which a conclusion can reasonably be drawn from the results. A lack of a meaningful comparative baseline can provide a threat by making it harder to produce a conclusion from the results without the relevant context. In order to provide descriptive statistics of the results, tasks have been repeated and average values have been used to compare against. Another possible threat may be provided by the lack of a formal hypothesis in the experiment. At the outset, two research questions have been provided and for each, a set of corresponding hypotheses have been constructed in order to aid in drawing a conclusion. To accompany these, non-parametric statistical tests have

been used to test the significance of the results gained. These tests make no assumption that the data is normally distributed and are suitable for ordinal data.

B. Discussion

In this paper an experiment was conducted to test a new fitness objective in the MultiRefactor tool. The refactoring coverage objective measures the extent of coverage of a program given by the set of refactorings in a refactoring solution and gives an ordinal score that indicates the number of elements refactored and also how many times. This objective drives a refactoring solution to increase the code coverage of the refactorings by looking at as many elements of the software as possible. It also reduces redundant refactorings applied to the software that reverse a previously applied change. The refactoring coverage objective was tested in conjunction with a quality objective (derived from previous experimentation) in a multi-objective setup. To measure the effectiveness of the refactoring coverage objective, the multi-objective approach is compared with a mono-objective approach using just the quality objective. The quality objective values are inspected to deduce whether improvements in quality can still be derived in this multi-objective approach. Then, the coverage scores are compared to measure whether the developed refactoring coverage function can be successful in improving the coverage of the refactoring approach and reducing redundant refactorings.

The average quality improvement scores were compared across six different open source inputs and, for all input programs, the mono-objective approach gave better improvements. The multi-objective approach gave improvements in quality across all the inputs. The average coverage scores were compared across the six inputs. The scores for the multi-objective approach were better in each case. Finally, the average execution times for each input were inspected and compared for each approach. The times for each approach were similar but, for most inputs, the mono-objective approach was quicker than the multi-objective counterpart. The times for each input program increased depending on the size of the program and the number of classes available. The average times ranged from two minutes for the Mango program, to 46 minutes for GanttProject.

In order to test the aims of the experiment and derive conclusions from the results a set of research questions were constructed. Each research question and their corresponding set of hypotheses looked at one of two aspects of the experiment. RQ1 was concerned with the effectiveness of the quality objective in the multi-objective setup. To address it, the quality improvement results were inspected to ensure that each run of the search yielded an improvement in quality. In all 30 of the different runs of the multi-objective approach, there was an improvement in the quality objective score, therefore rejecting the null hypothesis. RQ2 looked at the effectiveness of the refactoring coverage objective in comparison with a setup that did not use a function to measure the code coverage of refactorings. To address this, a non-parametric statistical test was used to decide whether the mono-objective and multi-objective data sets were significantly different. The coverage scores were compared for the multi-objective approach against the basic approach and the multi-objective coverage scores were found to be significantly higher than the mono-objective scores, rejecting the null hypothesis H20. Thus, the research questions addressed in this paper help to support the validity of the refactoring coverage objective in helping to improve the coverage of a refactoring solution in the MultiRefactor tool while in conjunction with another objective.

Of the tools proposed in previous research, there are two that use the RECODER framework to modify source code in Java programs. The CODe-Imp platform, developed by Moghadam and Ó Cinnéide [23], uses Abstract Syntax Trees to apply refactorings to a previously designed solution. It is outfitted with a selection of refactorings and metrics, and a number of search options, although there are no multi-objective search techniques available. Likewise, Trifu et al. [24] proposed an approach that uses RECODER to generate Abstract Syntax Trees. Their approach incorporates a number of tools to handle each stage. Their Inject/J tool is used to modify Java programs, with the help of RECODER. The MultiRefactor also uses RECODER in order to modify the Java code during the search. MultiRefactor, however, has a greater number of options than the other tools and a large amount of configurability. It has the ability to use both mono-objective and multi-objective approaches, and even has a many-objective genetic algorithm available. It also generates actual refactored, compiled code, in contrast to the majority of the refactoring approaches previously used. Many of these approaches only suggest refactoring sequences to be applied, and do not check the applicability of the refactorings. Although a number of refactoring studies experiment with different objectives to focus on in a software refactoring solution, none are concerned with the diversity of the code elements refactored. Therefore, the authors believe that this is a novel property to inspect when considering refactoring options in an automated maintenance setup.

For future work, further experimentation could be conducted to test the effectiveness of the refactoring coverage objective. The authors also plan to investigate other properties in order to create a better supported framework to allow developers to maintain software based on their preferences and their opinions of what factors are most important. It would also be useful to gauge the opinion of developers in industry and find out their opinion of the effectiveness of the MultiRefactor approach, and of the refactoring coverage objective in an industrial setting.

REFERENCES

[1] M. Harman and L. Tratt, "Pareto Optimal Search Based Refactoring At The Design Level," in 9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007., 2007, pp. 1106–1113.

[2] W. Mkaouer, M. Kessentini, S. Bechikh, K. Deb, and M. Ó Cinnéide, "High Dimensional Search-Based Software Engineering: Finding Tradeoffs Among 15 Objectives For Automating Software Refactoring Using NSGA-III," in Genetic and Evolutionary Computation Conference, GECCO 2014., 2014.

[3] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and M. S. Hamdi, "Improving Multi-Objective Code-Smells Correction Using Development History," J. Syst. Software., vol. 105, pp. 18–39, 2015.

[4] K. Deb and D. K. Saxena, "Searching For Pareto-Optimal Solutions Through Dimensionality Reduction For Certain Large-Dimensional

Multi-Objective Optimization Problems," in IEEE Congress on Evolutionary Computation, CEC 2006., 2006.
[5] H. K. Singh, A. Isaacs, and T. Ray, "A Pareto Corner Search Evolutionary Algorithm And Dimensionality Reduction In Many-Objective Optimization Problems," IEEE Trans. Evol. Comput., vol. 15, no. 4, pp. 539–556, 2011.
[6] F. Di Pierro, S.-T. Khu, and D. A. Savić, "An Investigation on Preference Order - Ranking Scheme For Multi Objective Evolutionary Optimization," IEEE Trans. Evol. Comput., vol. 11, no. 1, pp. 1–33, 2007.
[7] Q. Zhang and H. Li, "MOEA/D: A Multiobjective Evolutionary Algorithm Based on Decomposition," IEEE Trans. Evol. Comput., vol. 11, no. 6, pp. 712–731, 2007.
[8] K. Deb and H. Jain, "An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point Based Non-Dominated Sorting Approach, Part I: Solving Problems With Box Constraints," IEEE Trans. Evol. Comput., vol. 18, no. 4, pp. 1–23, 2013.
[9] L. Thiele, K. Miettinen, P. J. Korhonen, and J. Molina, "A Preference-Based Evolutionary Algorithm For Multi-Objective Optimization," Evol. Comput., vol. 17, no. 3, pp. 411–436, 2009.
[10] T. Van Belle and D. H. Ackley, "Code Factoring And The Evolution of Evolvability," in Genetic and Evolutionary Computation Conference, GECCO 2002., 2002, pp. 1383–1390.
[11] R. Vivanco and N. Pizzi, "Finding Effective Software Metrics To Classify Maintainability Using A Parallel Genetic Algorithm," in 6th Annual Conference on Genetic and Evolutionary Computation, GECCO 2004., 2004, pp. 1388–1399.
[12] A. D. Bakar, A. B. Sultan, H. Zulzalil, and J. Din, "Applying Evolution Programming Search Based Software Engineering (SBSE) In Selecting The Best Open Source Software Maintainability Metrics," in International Symposium on Computer Applications and Industrial Electronics, ISCAIE 2012., 2012, no. Iscaie, pp. 70–73.
[13] D. Fatiregun, M. Harman, and R. M. Hierons, "Evolving Transformation Sequences Using Genetic Algorithms," in 4th IEEE International Workshop on Source Code Analysis and Manipulation, SCAM 2004., 2004, pp. 65–74.
[14] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination of Refactorings For Improving The Class Structure of Object-Oriented Systems," in Genetic and Evolutionary Computation Conference, GECCO 2006., 2006, pp. 1909–1916.
[15] R. Lange and S. Mancoridis, "Using Code Metric Histograms And Genetic Algorithms To Perform Author Identification For Software Forensics," in 9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007., 2007, pp. 2082–2089.
[16] M. O'Keeffe and M. Ó Cinnéide, "Getting The Most From Search-Based Refactoring," in 9th Annual Conference on Genetic and Evolutionary Computation, GECCO 2007., 2007, pp. 1114–1120.
[17] M. O'Keeffe and M. Ó Cinnéide, "Search-Based Refactoring: An Empirical Study," J. Softw. Maint. Evol. Res. Pract., vol. 20, no. 5, pp. 1–23, 2008.
[18] A. C. Jensen and B. H. C. Cheng, "on The Use of Genetic Programming For Automated Refactoring And The Introduction of Design Patterns," in 12th Annual Conference on Genetic and Evolutionary Computation, GECCO 2010., 2010, pp. 1341–1348.
[19] M. Ó Cinnéide, "Automated Application of Design Patterns: A Refactoring Approach,", PhD theis, UC Dublin, 2000.
[20] M. Kessentini, W. Kessentini, H. Sahraoui, M. Boukadoum, and A. Ouni, "Design Defects Detection And Correction By Example," in IEEE International Conference on Software Engineering, ICSM 2011., 2011, pp. 81–90.
[21] M. Kessentini, W. Kessentini, and A. Erradi, "Example-Based Design Defects Detection And Correction," in 19th International Conference on Program Comprehension, ICPC 2011., 2011, pp. 1–32.
[22] M. Kessentini, R. Mahaouachi, and K. Ghedira, "What You Like In Design Use To Correct Bad-Smells," Softw. Qual. Journal., vol. 21, no. 4, pp. 551–571, Oct. 2012.
[23] D. R. White, J. Clark, J. Jacob, and S. Poulding, "Searching for Resource-Efficient Programs: Low-Power Pseudorandom Number Generators," in Genetic and Evolutionary Computation Conference, GECCO 2008., 2008, pp. 1775–1782.
[24] J. T. De Souza, C. L. Maia, F. G. De Freitas, and D. P. Coutinho, "The Human Competitiveness of Search Based Software Engineering," in 2nd International Symposium on Search-Based Software Engineering, SSBSE 2010., 2010, pp. 143–152.
[25] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "Search-Based Refactoring: Towards Semantics Preservation," in 28th IEEE International Conference on Software Maintenance, ICSM 2012., 2012, pp. 347–356.
[26] A. Ouni, M. Kessentini, and H. Sahraoui, "Search-Based Refactoring Using Recorded Code Changes," in European Conference on Software Maintenance and Reengineering, CSMR 2013., 2013, pp. 221–230.
[27] A. Ouni, M. Kessentini, H. Sahraoui, and M. S. Hamdi, "The Use of Development History In Software Refactoring Using A Multi-Objective Evolutionary Algorithm," in Genetic and Evolutionary Computation Conference, GECCO 2013., 2013, pp. 1461–1468.
[28] A. Ouni, M. Kessentini, H. Sahraoui, K. Inoue, and K. Deb, "Multi-Criteria Code Refactoring Using Search-Based Software Engineering: An Industrial Case Study," ACM Trans. Softw. Eng. Methodol., vol. 25, no. 3, 2016.
[29] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability Defects Detection And Correction: A Multi-Objective Approach," Autom. Softw. Eng., vol. 20, no. 1, pp. 47–79, 2013.
[30] A. Ouni, M. Kessentini, S. Bechikh, and H. Sahraoui, "Prioritizing Code-Smells Correction Tasks Using Chemical Reaction Optimization," Softw. Qual. Journal., vol. 23, no. 2, 2015.
[31] W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, and K. Deb, "Software Refactoring Under Uncertainty: A Robust Multi-Objective Approach," in Genetic and Evolutionary Computation Conference, GECCO 2014., 2014.
[32] M. W. Mkaouer, M. Kessentini, M. Ó Cinnéide, S. Hayashi, and K. Deb, "A Robust Multi-Objective Approach To Balance Severity And Importance of Refactoring Opportunities," Empir. Softw. Eng., 2016.
[33] M. W. Mkaouer, M. Kessentini, S. Bechikh, M. Ó Cinnéide, and K. Deb, "On The Use of Many Quality Attributes For Software Refactoring: A Many-Objective Search-Based Software Engineering Approach," Empir. Softw. Eng., 2015.
[34] W. Mkaouer, M. Kessentini, P. Kontchou, K. Deb, S. Bechikh, and A. Ouni, "Many-Objective Software Remodularization Using NSGA-III," ACM Trans. Softw. Eng. Methodol., vol. 24, no. 3, 2015.
[35] I. H. Moghadam and M. Ó Cinnéide, "Code-Imp: A Tool For Automated Search-Based Refactoring," in 4th Workshop on Refactoring Tools, WRT 2011., 2011, pp. 41–44.
[36] A. Trifu, O. Seng, and T. Genssler, "Automated Design Flaw Correction In Object-Oriented Systems," in 8th European Conference on Software Maintenance and Reengineering, CSMR 2004., 2004, pp. 174–183.
[37] M. Mohan, D. Greer, and P. McMullan, "Technical Debt Reduction Using Search Based Automated Refactoring," J. Syst. Software., vol. 120, pp. 183–194, 2016.
[38] E. Koc, N. Ersoy, A. Andac, Z. S. Camlidere, I. Cereci, and H. Kilic, "An Empirical Study About Search-Based Refactoring Using Alternative Multiple And Population-Based Search Techniques," in Computer and Information Sciences II., E. Gelenbe, R. Lent, and G. Sakellari, Eds. London: Springer London, 2012, pp. 59–66.
[39] I. Griffith, S. Wahl, and C. Izurieta, "TrueRefactor: An Automated Refactoring Tool To Improve Legacy System And Application Comprehensibility," in 24th International Conference on Computer Applications in Industry and Engineering, ISCA 2011., 2011.
[40] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A Fast And Elitist Multiobjective Genetic Algorithm: NSGA-II," IEEE Trans. Evol. Comput., vol. 6, no. 2, pp. 182–197, 2002.
[41] M. Fowler, Refactoring: Improving The Design of Existing Code. 1999.
[42] J. Bansiya and C. G. Davis, "A Hierarchical Model For Object-Oriented Design Quality Assessment," IEEE Trans. Softw. Eng., vol. 28, no. 1, pp. 4–17, 2002.
[43] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite For Object Oriented Design," IEEE Trans. Softw. Eng., vol. 20, no. 6, pp. 476–493, 1994