Assessing Single-Objective Performance Convergence and Time Complexity for Refactoring Detection

David Nader-Palacio  
Universidad Nacional de Colombia  
Bogota, Colombia  
danaderp@unal.edu.co

Daniel Rodríguez-Cárdenas  
Universidad del Rosario  
Bogota, Colombia  
danielh.rodriguez@urosario.edu.co

Jonatan Gomez*  
Universidad Nacional de Colombia  
Bogota, Colombia  
jgomezp@unal.edu.co

1 INTRODUCTION

Providing refactoring recommendations is a widespread practice that assists developers in ameliorating the maintainability and readability of their code [4]. Nonetheless, the procedure for automatically generating sequences composed of refactoring operations or the Refactoring Detection Problem (RDP) remains a complex [12] and arduous task for maintenance groups [13, 14].

Approaches that suggest refactoring operations must be very clear on how those refactorings are generated. Indeed, each approach must give a sound, concise and justified answer to the following questions: Which are the variables, the hyper-parameters, and the constraints of the model? How are the refactorings built and performed? Is the refactoring detection a truly multi-objective problem? Unfortunately, current research [16–18, 21, 22, 24–26] proposes informal optimization models for the RDP without proper performance convergence and time complexity analysis, making the experiments difficult to compare. The implementation and execution of refactorings from the reported models are ambiguous. For instance, Ouni et al.'s "DefectCorrection" algorithm did not clarify how the refactoring sequences were computed on the code to assemble final recommendations [23]. Therefore, performance convergence and time complexity analysis are required to empirically evaluate search-based techniques [5, 8].

In this paper, we employ the Artificial Refactoring GENERation (ARGen), an approach for detecting massive refactoring operation sets. ARGen allows researchers to estimate the impact of proposed changes to the source code before such changes are implemented. We analyze the performance and time complexity of ARGen with two baseline single optimization techniques and a Hybrid Adaptive Evolutionary Algorithm (HaEa) [8]. Eventually, the research is expected to contribute to the empirical analysis of the application of single-objective techniques in software maintenance problems.

2 A REPRODUCIBILITY ANALYSIS OF THE REFACTORING OPTIMIZATION MODELS

The reproducibility analysis - whose purpose is to manually verify the limitations of automatically performing software refactoring - consisted in assigning a value to each reproducibility characteristic (or dimension traced a conceptual matrix) for every reported model. The models were then ranked: zero points indicates the model was easy to reproduce; one or two points moderately hard to reproduce; three or more than three points hard to reproduce. If the behavior preservation dimension was unclear, the approach

---

ABSTRACT

The automatic detection of refactoring recommendations has been tackled in prior optimization studies involving bad code smells, semantic coherence and importance of classes; however, such studies informally addressed formalisms to standardize and replicate refactoring models. We propose to assess the refactoring detection by means of performance convergence and time complexity. Since the reported approaches are difficult to reproduce, we employ an Artificial Refactoring Generation (ARGen) as a formal and naive computational solution for the Refactoring Detection Problem. ARGen is able to detect massive refactoring’s sets in feasible areas of the search space. We used a refactoring formalization to adapt search techniques (Hill Climbing, Simulated Annealing and Hybrid Adaptive Evolutionary Algorithm) that assess the performance and complexity on three open software systems. Combinatorial techniques are limited in solving the Refactoring Detection Problem due to the relevance of developers’ criteria (human factor) when designing reconstructions. Without performance convergence and time complexity analysis, a software empirical analysis that utilizes search techniques is incomplete.

CCS CONCEPTS

- Mathematics of computing → Combinatorial optimization; Mathematical software performance; • Computer systems organization → Maintainability and maintenance;

KEYWORDS

Combinatorial Optimization, Mathematical Software Performance, Refactoring, Software Maintenance

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GECCO ’18 Companion, July 15–19, 2018, Kyoto, Japan  
© 2018 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.  
ACM ISBN 978-1-4503-5764-7/18/07...$15.00  
https://doi.org/10.1145/3205651.3208294

1 Regarding the impact on the research community and industry, all the artifacts of the project are available for any researcher or developer who requires performing refactoring analysis: https://github.com/danielrcardenas/argen.

2 Find the complete table in: https://argeneration.github.io/assets/pdfs/longtable.pdf
was labeled as hard to reproduce since the key characteristic of any refactoring is the ability to preserve the functionality. From Table 1, we conclude the search-based models fail in explaining the way of reproducing refactoring techniques (no evidence of a real refactoring outcome). Moreover, none of the papers perform algorithm analysis convergence, except for Seng, et al. [29].

3 REFACTORING DETECTION DESCRIPTION

Sets of classes, methods, fields and refactoring operations 3 are properly represented to be utilized in a combinatorial framework besides the metrics 4 utilized in the objective function. A formalism for refactoring is appropriates whether the research community desires to extend any technique, enclose other formalism in the framework (e.g., applying a hyper-heuristic or memetic algorithm for the software refactoring problem [3, 28]), or represent the refactorings in a data structure different from sequences (e.g., Directed Acyclic Graphs).

3.1 Formal Definitions

Object-oriented programming (OOP) is a software development style organized around objects, which can be used in paradigms like imperative, functional, or logic programming [19]. An object is defined as a run-time entity composed of 1) a state represented by Kleene star. Set $c = \{c \mid c \subseteq \mathcal{V}(c_a) \text{ and } |c| = k, \text{ where } k \text{ is the total number of System Under Analysis (SUA) classes and } c = \{c_a \in C\mid 1 \leq \alpha \leq k\}$.

Definition 3.1. (System Under Analysis definition). SUA is a software system composed of classes, methods, and attributes, where objects instantiate classes and communicate with each other through messages.

Definition 3.2. (Class definition). A class, prototype or template is a set of similar objects including collections of similar attributes and methods. In fact, classes are the programming-time counterparts of objects [27]. Assume a single class $c_{a}$ is a Cartesian product represented by $c_{a} = identifier \times Attribute(s) \times Method(s) = str \times str^{*} \times str^{*}$, where $str^*$ is a Kleene on string abstraction or chain. A set of classes is a power set of $c_{a}$, $C \subseteq \mathcal{V}(c_{a})$ and $|C| = k$, where $k$ is the number of objects in SUA classes and $c = \{c_{a} \in C\mid 1 \leq \alpha \leq k\}$.

Definition 3.3. (Method Definition). Methods are subroutines which define the set of valid messages the object may accept. Methods are represented by Kleene star. Set $M = str^{*} = \bigcup_{n=0}^{\infty}(str)^{n}$ and $\forall 1 \leq \alpha \leq k$ states $|M| = \beta_{a}$ is finite. So $M \subseteq \mathbb{N}$ and $M(c_{a}) = M_{a}$.

Definition 3.4. (Attribute Definition). Attributes (or fields) are sets of internal objects that represent a state. Attributes are represented by Kleene star. Set $A = str^{*} = \bigcup_{n=0}^{\infty}(str)^{n}$ and $\forall 1 \leq \alpha \leq k$ states $|A| = \gamma_{a}$ is finite. So $A \subseteq \mathbb{N}$ and $A(c_{a}) = A_{a}$.

Definition 3.5. (Refactoring Definition). The refactoring process consists in re-constructing the code design of a SUA without affecting the behavior functionality [4]. A refactoring is a function $R_{\delta} : \Theta \rightarrow \text{ (Code Modification) where } \delta$ represents a specific refactoring operation and $\Theta = c_{s} \times A_{s} \times M_{s} \times c_{t}$ is a Cartesian product parameter.

Refactoring properties are: 1) a solution set of all the possibilities of refactorings functions with refactoring parameters, coined as $R(\Theta)$, so $RI \in R(\Theta)$, is a specific solution or Refactoring Instance, 2) a refactoring recommendation $S_{i}$ is a set of Refactoring Instances $RI$ and a subset of $R(\Theta)$.

Definition 3.6. (Code Quality Metric Definition). A quality metric is a function $\eta_{i} : c_{a} \rightarrow \mathbb{R}$. Each class $c_{a}$ of the SUA has a set of metric values: $H_{a} = \{\eta_{j}(c_{a}), \eta_{j}(c_{a}), \ldots, \eta_{j}(c_{a})\}$, where $j$ is the total number of metrics and $H_{a} \in \mathbb{R}^{j}$.

We utilized Refactoring Impact PrEdiction (RIPE) to predict the impact of 12 refactoring operations on 11 code metrics [2] because our solution set of refactoring recommendation is too expensive to perform in a SUA, yet an estimation of the refactoring on the metrics can be computed. RIPE implements 89 impact prediction functions that show developers the variation of code metrics before the application of a refactoring operation. For instance: $LOC_{p}(c_{a}) = LOC_{p}(c_{a}) - LOC_{p}(m_{k})$ and $LOC_{p}(c_{a}) = LOC_{p}(c_{a}) + LOC_{p}(m_{k})$. For each Refactoring Operation $r_{a}$ and quality metric $\eta_{i}$ there is a prediction function $\text{Prediction}_{\delta, j}(c_{a}) = \tilde{\eta}_{\delta, j}(c_{a})$, which estimates the impact of the refactoring on the metric of a class (iff the refactoring affects the metric).

Definition 3.7. (Impacted Code Quality Metric Definition.) An impacted quality metric is a function $\tilde{\eta}_{\delta, j} : c_{a} \rightarrow \mathbb{R}$. Each class $c_{a}$, impacted by a refactoring operation $r_{a}$ of the SUA, has a set of metric values: $H_{a} = \{(\tilde{\eta}_{\delta, 1}(c_{a}), \tilde{\eta}_{\delta, 2}(c_{a}), \ldots, \tilde{\eta}_{\delta, j}(c_{a})\}$, where $j$ is the total number of metrics and $H_{a} \in \mathbb{R}^{j}$.

3.2 Refactoring Detection as a Combinatorial Problem

The Artificial Refactoring Generation (ARGen) considers all possible combinations of refactoring operations for a given SUA, which implies to operate in a large solution space. The purpose of the technique is to recommend massive artificial refactoring operations that fulfill the proposed parametric objective function.

The refactoring generation is a NP-Complete combinatorial problem 5. The solution constitutes a set of refactoring operations instead of a sequence of refactoring operations. The refactoring operations from the same set are independent of each other. ARGen does not guarantee generated outcomes reduce the error-proneness of an inspected piece of code (or SUA). Nevertheless, ARGen is able to explore the search space in actionable areas, if and only if, the developers/researchers properly tune the objective function based on their interest in SUA.

3.2.1 Characterization of the Search Space. Figure 1 depicts the search space of the Artificial Refactoring Generation. The feasible region represents refactoring that fulfill defined constraints of the proposed objective function. The actionable region represents refactoring solutions that reduce error-proneness. The search space increases by the expression $(C^{2} + A + M)^{r}$, where $r$ is the number of refactoring operations in a set (e.g., a system with 10 classes, 10 attributes, and 10 methods would have a size of 10,000 if the set is composed of one refactoring). Our paper demonstrates that

\footnotesize{Find the list of refactoring operations in: https://argeneration.github.io/assets/pdfs/RefReFactors.pdf
Find the list of metrics in: https://argeneration.github.io/assets/pdfs/RefMetrics.pdf

\footnotesize{The ARGen was demonstrated through the Subset Sum problem computational complexity analysis (Subset-Sum Problem = ARGen)}
the number of classes involved in the optimization affects the time complexity.

3.2.2 Objective Function. The objective function represents a ratio that measures the normalized difference between the actual metrics and the impacted metrics of the system under analysis. The inputs constitute a set of refactoring operations composed of real objects from the system (i.e., target class, source class, fields, and methods) and a weighted vector \( w \) that captures the relevance of each metric.

**Definition 3.8. (Code Quality Additive Metric Definition).** The use of quality metrics seeks not only to identify refactoring opportunities but also to capture developers’ interest. This formula computes

\[
\Upsilon_{RI}(j) = \sum_{i=1}^{N} \eta_j(c_{si})
\]

a general value for one exact metric on the processed SUA.

The parameter \( \Phi_j \) of the impacted sum function \( \Upsilon_{RI}(\Phi_j) \) is composed of a metric \( j \in J \) and the solution set \( S_i : \Phi_j = (j, S_i) = (j, (R1, R12, ..., R1i, ..., R1T)) \), where \( T \) is the total number of Refactoring Instances in \( S_i \).
The following formula estimates a general value for one specific metric on the processed SUA according to a solution set.

\[ Y_{HF}(\Phi_j) = \sum_{i=1}^{n} \sum_{a=1}^{m} \max_{1 \leq j \leq H_a} \left( \frac{7\Phi_j(c_{ia})}{\bar{\eta}_{j}(c_{ia})} \right) \]

where \( \Phi \) or vector of weights are numbers between \((-1, 1)\) and \(\rho(\Phi)\) penalization. The vector of weights is the formalism where developers convey their interest in metrics to obtain meaningful suggestions of classes to be reconstructed [1]. When the numerator is less than the denominator, the predicted system improves its quality metrics. In other words, if the objective function is an improper fraction, then the predicted system is worse (in terms of quality) than the actual. The input of the objective function is a set \( \Phi \) of refactoring recommendations; alternatively, an output is a value that represents the estimation of the metrics after applying the refactorings operations in \( \Phi \).

The objective function does not ensure refactoring opportunities in a SUA [1], yet an estimation of how the refactorings are impacted by the change of a quality metric [2]. The implementation of the objective function (Figure 2) required the design of computational techniques to adapt the Artificial Refactoring Generation and the definition of specific refactoring data structures (see figure 6 from the web page).

3.2.3 Refactoring Constraints. We pose a catalog of constraints that depend on the refactoring operation structure and the general object-oriented guidelines. Figure 3 depicts a typical configuration of a solution after applying a mutation, a variation’s operator, or a new generation. Consider a class \( C1 \) a subclass of \( C2 \). If we try to apply a "Pull Up Field" on a solution where \( C1 \) constitutes the source and \( C2 \) the target, then that operation violates the constraint \( \text{SRCSubClassTGT} \).

\[ \text{Definition 3.11. (The Objective Function Definition).} \]

The objective function is a parametric optimization function receives developer-defined goals. We use min-max normalization for BQR to put the metrics in a positive scale:

\[ \text{Obj}(\Phi) = \sum_{j=1}^{l} \left( \frac{\sum_{j=1}^{l} \left( \frac{Y_{HF}(\Phi_j) - \min(Y_{HF}(\Phi_j))}{\max(Y_{HF}(\Phi_j)) - \min(Y_{HF}(\Phi_j))} \right)}{\sum_{j=1}^{l} \left( \frac{Y_{HF}(\eta_j) - \min(Y_{HF}(\eta_j))}{\max(Y_{HF}(\eta_j)) - \min(Y_{HF}(\eta_j))} \right)} \right) + \rho(\Phi) \]

where \( w \) or vector of weights are numbers between \((-1, 1)\) and \(\rho(\Phi)\) penalization. The vector of weights is the formalism where developers convey their interest in metrics to obtain meaningful suggestions of classes to be reconstructed [1]. When the numerator is less than the denominator, the predicted system improves its quality metrics. In other words, if the objective function is an improper fraction, then the predicted system is worse (in terms of quality) than the actual. The input of the objective function is a set \( \Phi \) of refactoring recommendations; alternatively, an output is a value that represents the estimation of the metrics after applying the refactorings operations in \( \Phi \).
Assessing Single-Objective Performance Convergence and Time Complexity for Refactoring Detection

code of this approach is available online. The Hybrid Adaptive approach utilizes the repairing functions and six different genetic operators (Figure 4) that were specially made for exploring in spaces where the algorithm preserves the object-oriented structure.

Figure 4: HaEa with invariable (fixed) chromosome uses: Refactoring Operation Mutation that changes a complete refactoring operation label preserving its internal parameters (src, tgt, fid and mtd); Refactoring Operation Class Transposition that interchanges the src with tgt parameters inside a specific gen (the gens is selected by following a Gaussian distribution); Refactoring Operation Crossover that crosses two refactoring operations in a random position (Gaussian distribution) of the chromosomes. HaEa with variable chromosome uses: Adding Refactoring Operation that adds a specific number of refactorings to the individual; Deleting Refactoring Operation that deletes a specific number of refactorings to the individual; and Joining Refactoring that mixes two individuals into one.

4 EVALUATION

We evaluate the performance of the approach when generating massive artificial refactoring recommendations and to present the results of baseline search techniques through proper statistical analysis. We performed a preliminary experiment and a formal experiment from previous reports of automated refactoring [18, 29]. The preliminary experiment evaluated the de--ability and the formal experiment validated the performance of the models and the time complexity given a number of classes.

We employed three open software systems, which have been regularly reported in evolutionary computation approaches for refactoring [11, 12, 18, 26]: Commons Codec v.1.10 with 123 number of classes (CCODEC), Acra v.4.6.0 with 59 classes (ACRA) and JFreeChart v.1.0.9. with 558 classes. Two datasets were initially configured for algorithms performance analysis. The first dataset contains all source classes from CCODEC and the second dataset all source classes of ACRA. In addition, we employed 6 datasets from JFreeChart (classes inside each dataset are hierarchy related). These datasets were applied for time complexity analysis.

In the preliminary results, the Shapiro-Wilk test suggested the data featured no normal distribution for both datasets ACRA and CCODEC. The \( p \) - values constitute less than 0.05 so that the alternative hypothesis is rejected. Hill Climbing and Simulated Annealing were compared against HaEa with a Wilcoxon test. The null hypothesis \( H_0 \) states that the median of Hill Climbing fitness is the same as HaEa’s and the alternative hypothesis (\( H_1 \)) constitutes that the median of HaEa fitness is greater than the baseline algorithms. The \( \alpha \) value follows 0.05 level of significance.

Each dataset generated a sample of fitness values that were organized sequentially. The actual position was compared respect to the previous one by keeping the least value in each evaluation (steady state). Since all the experiments were executed 30 times, we organized the results in a matrix where rows constitute the evaluation (fitness or time) and the columns the experiments. The median, the median deviation, maximum and minimum values for each row were calculated. As far as the formal experiments are concerned, Experiment I. Algorithm’s Performance seeks to validate which approach conveys the best fitness behavior results and Experiment II. Time Complexity of the recommendations assesses the relationship between the number of classes and the time spent in generating artificial refactorings for such datasets.

4.1 Performance Convergence Experiment I

The following research question narrowed the study: RQ1. What is the technique’s performance when generating feasible recommendations? To answer RQ1, first, the source code datasets were organized according to the number of classes and the second baseline algorithms for convergence assessment were applied to the datasets.

4.1.1 Algorithm’s Performance Results. We compared three curves that represent computed fitness values for HC and SA (best, median and worst). Figure 5 depicts for three algorithms the largest evaluations attempted during our experiment. The best fitness rates in (a) and (b) are relatively homogeneous during evaluations. In (a) the median and worst values became stable after 3,000 evaluations, however, in (b) the median values are variable though all evaluations (0.97775 ± 0.00633[5000], 0.97386 ± 0.00631[7000], 0.97167 ± 0.00354[10000]). We observed that HaEa (c) obtained the best values starting from 6,000 evaluations for HC (\( p \)-value = 0.00346) and 3,000 evaluations for SA (\( p \)-value = 0.04117). By 10,000 evaluations, HaEa values are significantly lower than HC (\( p \)-value = 0.00016) and SA (\( p \)-value = 0.00014). Although, HaEa behavior (c) in large evaluations is better in all sample points (Table 2) with a remarkable \( p \)-value = 0.00023. Our results exhibit that HaEa presents good performance in large evaluations. HC and SA experience relative better results in few steps than HaEa, even though the evolutionary approach reaches the best values after certain evaluations. Previous studies pose combinatorial representations of refactorings [6, 10, 20, 29], but this paper establishes a benchmark problem contrasting baseline algorithms (HC and SA) versus HaEa with an extensive performance study.

Table 2: Performance Rates over Number of Evaluations for HC, SA and HaEa in Acra System

<table>
<thead>
<tr>
<th>Evaluation</th>
<th>Hill Climbing</th>
<th>Simulated Annealing</th>
<th>HaEa</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.97589</td>
<td>0.97145</td>
<td>0.96543</td>
</tr>
<tr>
<td>20000</td>
<td>0.97529</td>
<td>0.9695</td>
<td>0.96133</td>
</tr>
<tr>
<td>40000</td>
<td>0.97077</td>
<td>0.9684</td>
<td>0.95623</td>
</tr>
<tr>
<td>50000</td>
<td>0.96862</td>
<td>0.9684</td>
<td>0.9557</td>
</tr>
<tr>
<td>60000</td>
<td>0.96815</td>
<td>0.96819</td>
<td>0.95493</td>
</tr>
</tbody>
</table>

\( p \)-value = 0.00023 vs HaEa group.

The following plots Figure 6a and Figure 6b aim to compare four algorithms. In system CCODEC, the best behavior corresponds to...
Simulated Annealing algorithm for 1000 evaluations, yet it presents the highest variability. While, in ACRA, HaEa experienced the best performance for 10000 and 60000 evaluations. HaEa presents good performance in large evaluations is supported by box-plot analysis. Ten thousand evaluations were executed on ACRA system, only HaEa obtains the best median value. Likewise, after 6,000 evaluations, HaEa was still searching for fewer values. Hill Climbing and Simulated Annealing induced better results in few evaluations like in CCODEC experiments with 1000 and 2000 evaluations respectively.

4.2 Time Complexity Experiment II

The JFreeChart datasets were employed to perform the time complexity evaluation. The following research question narrowed the study: **RQ2. How is the relation between the variables \( n \) classes and \( t \) time?** To answer RQ2, first, the source code datasets were organized according to design relationships and heritage and, second, the time analysis included both baseline and evolutionary techniques to establish the trend line between classes and time.

4.2.1 Time Complexity Results. Figure 7 depicts a comparison of the mean time complexity between baseline algorithms and HaEa search techniques for 30 independent experiments. The time complexity points forms an exponential patterns (HaEa with an equation model \( y = 470.537e^{(0.024x)} \)).

The percentage increase in HC and SA techniques from 11 classes to 71 was 504.41% and 515.92% compared to 419.63% in HaEa’s. The results show that each time the experiment increased the number of the classes, baseline algorithms took more time processing classes for refactoring than HaEa. HaEa seems to be a better model to estimate refactoring recommendations in less time.

5 DISCUSSION

The following listing is a real JSON output from ACRA generated by ARGgen, which recommends two types of refactorings. Such output allowed us to manually implement the refactoring.9 Understanding software design artifacts are not enough for detecting refactoring regardless the size of the SUA. ARGgen widely explores the search space to detect refactorings without any previous knowledge about the system. An automatic refactoring detection approach by employing search techniques is an open research problem. However, this research considers the single-objective case and the main goal embodies to correctly insert the assessment techniques (convergence and time complexity) into the empirical studies and not to solve the RDP.

Our main finding that the combinatorial techniques are validated in terms of performance and time complexity is supported by the experiments performed in section 4. Answering RQ1, for dataset CCODEC, three defined curves (best, median and worst) diverge for HC and SA after 2,000 evaluations; nonetheless, for ACRA, both algorithms converged in a mean value of 0.96819 with a standard deviation of 0.00174 after 60,000 fitness evaluations. Conversely, answering RQ2, the relation between a number of classes and time is exponential for both HC and SA.

Seng, et al. [29] research contains the most relevant convergence study to compare with. They proposed an evolutionary algorithm for optimizing class structure in an open source system to keep class level refactoring. The outcome is a refactoring developed by the value of several quality metrics and the number of violations of object-oriented design guidelines. Seng’s study design considered only one refactoring (move method) and restricted number of classes; thus, the search space was significantly reduced. Whereas, ARGgen explores all the possible search space for the given SUA with 12 refactoring operations. Ignoring convergence and time complexity in software empirical analysis implies that search-based approaches do not guarantee a reasonable number of evaluations to obtain proper solutions on time. We summarize the limitations of this research as below: our research did not develop a refactoring

Assessing Single-Objective Performance Convergence and Time Complexity for Refactoring Detection

ARGen does not assist developers in what software properties need to change (e.g., reduce cyclomatic complexity by 10% and increase cohesion by 0.2 in class C), but how to change the software (e.g., a Replace Method with Method Object is a suggested refactoring operation for this source “src” and target “tgt” classes according to some fixed weights in the objective function). The generated sets of refactorings do not exclusively represent actionable recommendations, this depends upon developers’ criteria; nonetheless, the sets do represent feasible individuals that fulfill fitness parameters and object-oriented guidelines. Our approach does not analyze the convergence-complexity limitations of multi-objective optimization approaches, however, interpreting the results of the reproducibility section, we intuit that RDP based on quality metrics can be handled without multi-objective optimization. Finally, the empirical evaluations concentrated on algorithm performance and refactoring structure coherence.

Regardless of reported empirical studies [15, 16, 18, 21, 23, 24, 26], which exhibits precision and recall measures, the refactoring process highly depends on human factor that comprises specific domain knowledge and expertise when designing reconstructions on the code. Furthermore, the proposed set theory refactoring formalization is a first step to envision the refactoring recommendation as a graph instead of a sequence [30]. A graph representation of refactorings would be powerful and flexible.

We, indeed, accomplished to generate massive artificial refactorings, yet we cannot guarantee that none of those refactorings were actionable. Actionability implies not only developers’ interest in quality metric but also developers’ criteria (how to design a SUA) that should be extracted from their minds. Consequently, search techniques or combinatorial analysis are insufficient approximations to tackle the RDP because those techniques cannot acknowledge how the developer’s mind is simulated, employed, or mapped to make the exact context (human design process) for recommending refactorings. Moreover, notice that assessing convergence requires a large number of evaluations and the time complexity of applying search techniques in just detecting a refactoring—with as possible fewer constraints—is exponential.

```
    "M": [  
      {  
        "M": [  
          {  
            "tgt": "[TypeDeclaration [name=NewCrashReporterPersister]]",
            "src": "[TypeDeclaration [name=C拉斯ReporterPersister]]",
            "mtd": "[Store|EXISTS]"
          }
        ]
      }
    ]
```
6 CONCLUSION AND FUTURE WORK

We introduce a systematic formal approach to generate artificial refactoring recommendations based on quality metrics, which is used for assessing convergence and time complexity. We contributed to reducing the gap between the software empirical analysis (in the context of refactoring detection process) and search based techniques by establishing the fundamentals of performance convergence and time complexity in a unified mathematical theory toward a combinatorial optimization model. Our paper propounded a definition, an implementation, and an evaluation of a systematic approach that empirically analyses the convergence of a set of artificial refactoring operations. We plan to research on how our approach automatically adjusts the objective function weights to reduce the error-proneness in order to avoid multi-objective optimization since the problem does not exhibit strongly opposite objectives.

ACKNOWLEDGMENTS

The authors are grateful to Carlos Sierra, Oscar Chaparro, and Miguel Ramirez for their valuable comments and helpful suggestions during the model implementation and statistical analysis. The authors would also like to thank the former members of ALIFE and SEMERU research groups and the anonymous reviewers for their valuable feedback.

REFERENCES


