Research Statement

Morteza Zakeri-Nasrabadi, Ph.D.

June 2023

"Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it."

✠ Alan Perlis (1922—1990)

1 Overview

My area of interest is automated and intelligent software engineering (AISE) with the goal of making software development and maintenance effective, efficient, and economical. My research has focused on developing automated systems and tools for software quality measurement and improvement, test suite generation and minimization, technical debt estimation and reduction, and code recommendation. The main systems I have developed include a testability prediction system [1]–[3] that uses machine learning models to predict the testability of the classes and systems under test; an automated refactoring engine [4] that automates the application of 18 refactoring operations at the source code level to improve eight software quality attribute; and a format-aware fuzzer [5] that generates test data for software applications with complex input formats, such as PDF readers. The main techniques underlying the proposed systems are white-box compilers (compiler II), model-driven translation, machine-learning algorithms (software II), and evolutionary computing. The proposed tools and systems significantly advance the state of the art in the automated software engineering field and enable new paradigms for improving the quality of real-life software systems. As one of the interesting findings in automatic paradigms, my research shows that shallow learning in some code-related tasks, such as method name recommendation, is more effective and efficient than deep learning [6]. The rest of this research statement discusses my research motivation and goals, summarizes the previous and present of my research during my M.Sc. and Ph.D., and outlines the future directions of the research that I intend to pursue.

2 Motivation, importance, and goals of my research

Software engineering (SE) is intricate because software systems and ecosystems are inherently complex, intangible, and unpredictable. Software engineers must deal with non-trivial activities such as testing, quality assurance, and maintenance during the software development life cycle (SDLC). Efficient automation of these activities not only increases the process quality and product reliability but also leads to economic savings. Unfortunately, in practice, search-based software engineering (SBSE) techniques fail to automate emerging tasks, such as measuring and improving software quality, test data generation for complex format inputs, and code recommendation. I use machine learning software engineering (MLSE) besides SBSE to address open problems in automating SDLC activities, including quality prediction, code suggestion, refactoring, testing, debugging, repair, and maintenance.

Automation approaches must deal with a variety of software development artifacts, *e.g.*, requirements, production codes, test codes, and design diagrams, requiring different preprocessing and artifact representation mechanisms. However, most research works have been dedicated to automating code-related tasks. I have found that the automation of the majority of these activities can be achieved through the effective representation of

program source codes and in general a proper artifact similarity measurement instrument. It is highly important to uncover such common aspects by interpreting newly developed approaches to improve the generalization and applicability of proposed solutions and provide a solid theoretical framework for automated software engineering.

My research aims to provide intelligent and off-the-shelf methods and tools for software engineers to build their software systems with the highest quality, lowest cost, and minimum time. These tools are expected to be executed on developers' computer systems with low latency and overhead. My long-term goal is to generalize the proposed automation SE tools to work on different programming languages, various software projects' instances, and real-world and large-scale software applications, with good accuracy and performance.

During my graduate course, I had the chance to collaborate with researchers worldwide who are pioneers in the field and have similar research interests, in addition to my supervisor and advisor, to realize some aforementioned goals. I collaborated with Dr. Fabio Palmoba (University of Salerno), Dr. Mohamed Wiem Mkaouer (Rochester Institute of Technology), Prof. Chanchal K. Roy (University of Saskatchewan), and Prof. Alexander Chatzigeorgiou (University of Macedonia). I am highly motivated to pursue and extend such collaborations to advance the AISE field and expand its benefits to the community.

3 Research in automated and intelligent software engineering

During the course of my M.Sc., I applied deep learning techniques to learn the structure of complex file formats grammar and build a generative model to generate new test data improving the effectiveness of file format fuzzers. Regardless of inventing sophisticated test data generation methods, it was found that the test coverage is still very low in some of the software under test (SUT) due to software testability issues. Indeed, the low testability of the SUT leads to increasing test effort while decreasing test coverage. Therefore, in my Ph.D., I decided to focus on measuring and improving the testability of software systems artifacts to prepare the artifacts for testing before using automatic testing tools on a given software. The following summarizes the technical aspects of my research work during my M.Sc. and Ph.D. courses.

Deep learning-based test data generation with DeepFuzz Generating test data for complex input files, such as PDF, with the ability to achieve high code coverage and find more probably existing bugs, requires that test data pass all of the parsing stages of the SUT. Randomly generating such test data lead to very low code coverage which cannot guarantee the absence of bugs and the reliability of software. On the other hand, generating test data from grammar requires extracting the grammar or model of the file manually, which is expensive and time-consuming. I proposed DeepFuzz [5], a fuzzer that utilizes a deep neural language model to learn the structure of complex input formats. The learned model is then sampled with various strategies to generate new fuzzed files as test data. The first problem we encountered was finding a mechanism to distinguish between data and meta-data. To do so, we applied a reasonable trick: As meta-data repeated in almost every sample of a file format, the learned model predicts the meta-data with higher probability in comparison to the data. By putting a threshold at a model output, the data and metadata are distinguishable.

Aims to seek bugs in software by fuzzing techniques, the second problem was how to determine which byte should be fuzzed to reveal failures in the SUT. This can be done by targeting different stages of input processing that SUT used them. For example, if we would like to fuzz the parser, we should fuzz meta-data because the parser usually deals with meta-data to validate the format of the input file and extract data. On the other hand, if we would like to fuzz the execution stage, it may be better to fuzz data. The ability of the learned model in distinguishing data and meta-data is used to determine the place of fuzz in the file. In addition to determining the place of the fuzz, we should inform the value for replacing the third problem we addressed. As we know, the goal of fuzzing is creating the malformed input, and hence the most inappropriate byte is expected to be replaced in the input file. Again most inappropriate byte can be determined by the learned neural language model. It is enough to select a byte with the lowest likelihood instead of the highest likelihood used in the default manner.

Contributions. DeepFuzz [5] significantly outperforms previous file format fuzzer, such as AFL and Augmented-AFL, in terms of the achieved code coverage level for the SUT with complex input formats, specifically, PDF readers.

Learning to predict test effectiveness with Coverageability Predicting test effectiveness is vital since finding all faults in SUT is not a decidable problem. Test effectiveness depends on the extent to which the selected test cases cover the SUT which is only computable after testing the SUT. For this reason, measuring test coverage is expensive and time-consuming, requiring executable production code, installed dependencies, available resources, etc. I introduced a new quality attribute, Coverageability, to describe the test effectiveness based on test coverage and a machine-learning regression model, to estimate the Coverageability of each unit in the program under test [2]. Coverageability is the extent to which a given unit under test, *e.g.*, a class is expected to be covered by a test case.

There are three main challenges while adopting machine learning to build a Coverageability prediction model. The first challenge is selecting metrics to build the feature space. Initially, 54 known software metrics were collected. The feature space was extended by adding statistical and lexical metrics to obtain better performance. The second challenge is to provide an adequate number of samples to train and test the machine learning model. We computed the actual value of Coverageability using the size and the geometric average of statement, branch, and mutation coverage of the test suite generated by running EvoSuite on 23000 Java classes. Finally, the third challenge is selecting the most appropriate combination of test adequacy criteria, representing test effectiveness. Our experiments show that combining coverage criteria provides a relatively more accurate indication of test effectiveness than using only a single criterion. In summary, Coverageability as a new quality attribute can best be used to estimate the test effectiveness of SUT just in time and developers can continuously change their code in such a way that achieves the maximum Coverageability.

Contributions. The proposed Coverageability prediction models [2] have improved the MAE by 5.78% and R^2 score by 20.71% compared with the state-of-the-art approach for test effectiveness measurement, which only relies on branch coverage.

Software testability prediction with ADAFEST Despite a large number of studies on software testability, we observed that the relationship between testability and test adequacy criteria had not been explored empirically, and testability metrics still are far from measuring the actual testing effort. I combined Coverageability with test effort (*i.e.*, time budget and size of the test suite) to formalize a mathematical model for measuring software testability based on ISO/IEC 25010:2011 standard [3]. Again a machine learning model was trained on a large corpus of already tested software projects to build a testability prediction model for just-in-time evaluation of the SUT testability. Connecting runtime information to the static properties of the program throughout the learning-based algorithms is a key point in measuring software quality, specifically testability. Using machine learning interpretation techniques, mainly permutation importance analysis, my research automatically designated the most influential source code metrics affecting the SUT testability.

The proposed ADAFEST [3], [7] tool can be used to predict the testability of SUT without any need to program execution. Our experiments revealed that applying automated refactoring operations significantly improves SUT testability and the testability prediction model guides developers to enhance the testability of their classes while coding. Previous studies on improving software testability with automated refactoring have yielded inconclusive results. Investigating the impact of automated refactoring on testability is crucial since refactoring is a kind of program transformation that, unlike testability transformation techniques preserves behavior and produces a permanent and usable version of the software. Moreover, testing is a non-trivial and resource-consuming software engineering activity that entirely depends on system testability. It is possible to extend to proposed testability prediction model to work at the design level, where not all source code metrics are available and other artifacts such as class and package diagrams must be targeted for the testability measurement and improvement.

Contributions. AFASET [3] is the first tool that measures testability based on the ISO/IEC 25010:2011 standard with the help of intelligent models. It enables the automatic determination of important factors affecting software testability in practice.

Flipped boosting of automatic test data generation tools with CodART Soon after I developed the first testability prediction model and find the important metrics affecting testability, I realized that it is possible to improve the performance of automatic test data generation tools, such as DeepFuzz [5] and EvoSuite, without directly changing their internal structure and algorithms. The idea was to maximize the testability of a SUT by finding the most appropriate refactoring sequence and applying that refactoring sequence to SUT before testing it using test data generation tools. Testability prediction mode can be used as a fitness function in search-based software refactoring methods to find the refactoring sequences maximizing testability. This way the performance of test data generation tools regarding test adequacy criteria is improved since the algorithms used in these tools already work fine on SUT with an acceptable degree of testability. The main challenge was that despite many studies on search-based software refactoring, there were only a few tools that operate at the source code level. Most search-based refactoring studies use a simplified intermediate model extracted from the source code to work on. Moreover, the implementation of many automated refactoring papers is not available making their usage questionable and complicated.

To address the above problems, I designed and developed CodART [4], an automated refactoring toolkit that is capable to identify and apply 18 refactoring operations at the source code level. The proposed refactoring engine can be used to refactor Java programs. However, it is modular, well-documented, and provides API to add support for other programming languages and refactoring. CodART uses ANTLR which generates the program parse tree and provides the depth-first search (DFS) algorithm to visit parse tree nodes along with a call-back mechanism that calls a specific method when the visiting algorithm is entered or exited from the node. Using this mechanism, CodART finds the locations which should be modified for a given refactoring operation. In addition, call-back functions are attached to grammar rules to improve performance.

CodART improves the testability of Java projects by an average of 29.95%, provided that a relevant and precise testability prediction objective is used during refactoring sequence optimization. As a result, the performance of automated test data generation tools such as EvoSuite increased by 16.08% and 15.24% for branch and statement coverage. I called this approach *flipped or reverse boosting of automatic test data generation tools* in which the performance of automatic test data generator improved without directly modifying the tool. CodART also takes care of seven other quality attributes, including modularity and QMOOD attributes due to using the NSGA-III many-objective optimization algorithm. CodART has demonstrated a quality gain of 20.06%. Moreover, the manual precision of the suggested refactoring solutions has improved by 19.5% compared to state-of-the-art refactoring tools. The proposed flipped boosting mechanism can be generalized to use in other areas of automated software engineering, such as automatic software fault localization. The thinking paradigm behind flipped boosting mechanism is appreciated in a way that instead of trying to build a system for solving a sophisticated problem, try to make a simplified version of the problem such that it can be solved with existing solvers.

Contributions. CodART [4] is the first automatic refactoring engine that considers testability improvement while caring about human-centric software quality attributes, such as reusability and understandability.

Software entity name suggestion apparatus (SENSA) One of the main prerequisites of successful program transformation with refactoring from a development viewpoint is to have a transformed program with the appropriate names for newly created entities, mainly classes, and methods. Refactoring operations such as extract classes and extract methods create new classes and methods which require a proper name. State-of-the-art naming techniques use deep learning to compute the methods' similarity considering their textual contents. They highly depend on identifiers' names and do not compute semantical interrelations among methods' instructions. Source code metrics compute such semantical interrelations. I have designed and developed SENSA [6], [8], a tool that uses using source code metrics and a K-nearest neighbor learning algorithm to measure semantical and

structural cross-project similarities for recommending method names. SENSA achieves a precision of 65.49% with a recall of 59.15% on 420528 Java methods which is the highest score among state-of-the-art approaches in this field. In addition, the average response time of method recommendation is 0.52 seconds which is lower than deep learning-based approaches. Finally, the SENSA is applicable to used for programming languages other than Java, mainly C# and C++. It achieves an F1 score of 35.91% on predicting C++ method names with no change in the dataset and methodology, indicating its language neutrality. Overall, my research shows that a shallow learning approach based on source code metrics achieves greater efficiency and effectiveness in method name recommendations than deep learning approaches.

Contributions. SENSA [6] is a lightweight software entity name recommendation tool that significantly outperforms state-of-the-art deep-learning-based tools in terms of efficiency and effectiveness.

Research in progress For the time being, I am working on ADAFEST, CodART, and SENSA to complete their implementations, fix their bugs, and published their relevant papers. I am enhancing the ADAFEST testability prediction model to work at design levels where the source code of the software system is not available or developed. I am also adding new refactoring operations to CodART, including our recent Extract Method and Move Method batch refactoring approach based on advanced graph analysis [9] and refactoring to creational design patterns highly related to software testability, including Factory Method and Dependency Injection. This way the support for testability measurement and improvement of software artifacts at the design stage, *e.g.*, class diagram, are provided by the ADAFEST and CodART tools. It is straightforward to extend SENSA for supporting class name recommendations based on source code metrics and shallow learning algorithms. Such a model also can be used for similar code-related tasks, such as fault prediction and code smell detection. I am adding the support for class name prediction to ADAFEST. Finally, I am adapting the ADAFEST testability measurement model to work with software requirements artifacts using advanced natural language processing techniques.

4 Future directions of my research

My research plans for future work center around my long-term career goal to create generalized intelligent systems for automated software engineering. I briefly outline some important directions for my future research in the area of the AISE field. There are vast opportunities regarding the topics discussed in my previous and current research, most importantly those concerning downstream tasks on source code, automated refactoring performance, and reverse boosting frameworks. I think that what propose in the following highly helps to bridge the gap between researchers and practitioners in the field.

A solid automated software engineering framework based on source code similarity measurement I believe that code similarity measurement can be used as a fundamental component for all data-centric solutions in software engineering. The general theory of code similarity enables the automation of many laborious programming tasks, including but not limited to automatic clone detection, code recommendation, defect prediction, smell detection, vulnerability finding, refactoring recommendation, and quality measurements. Although researchers developed automated approaches separately in each domain, the relationship between most of these applications and code similarity measurement as a common principle has not been well studied. Investigating the common aspects of the aforementioned applications with respect to code similarity measurement help to develop versatile tools that can perform multiple tasks efficiently. A powerful code recommendation tool can be designed and developed to make a super agile and automated software development methodology. Achieving this final goal requires attention to some remaining uncovered aspects in source code engineering, described in the following.

Combining deep and shallow learning for code-related tasks Data-driven approaches for automating software engineering tasks require appropriate artifact representation, e.g., source code, design, and requirements representations to extract features and be fed into statistical and learning algorithms. The benefit of deep learning methods on feature extraction from source code is obvious. They remove the manual feature engineering and provide dense and semantical source code representations. Large language models (LLMs), such as BERT and GPT, are appropriate systems for automating multiple code-related tasks. However, they require many computational resources, large and quality datasets, and fine-tuning effort to make sound full results. Moreover, deep models are difficult to interpret and explain. Metrics-based source code representation, on the other hand, provides a lightweight, interpretable, language-independent, and efficient method to represent source codes as a fixed-length vector. I plan to explore the combination of manual and automatic source code feature engineering methods and their impact on my proposed tools. There are several ways to combine two vectors, including concatenation, linear combination, merging, etc. Therefore, investigating the best combination methods is an open problem.

Connecting code smell detection and automated refactoring research Despite many attempts to propose automated systems for code smell detection and refactoring there are theoretical and practical gaps between code smell detection tools and automated refactoring tools. Most search-based refactoring approaches use a random initialization strategy, in which the refactoring operations are selected and parameterized randomly leading to the explosion of search space and missing optimized solutions. The generation of the initial population is expected to be based on smelly entities to avoid search space explosion and make the search process efficient and precise. A missing part in automated refactoring research is the identification of refactoring opportunities such that they can fully parameterize the refactoring application function. Identifying refactoring opportunities in object-oriented code is a critical stage that precedes the actual refactoring process. It requires code smell detection for many types of smells. My recent systematic literature review on code smell datasets [10] indicates that there is not any dataset for six of the smells discussed by Fowler and Beck. I have also noticed that the identification of most refactoring activities (52 out of 72) proposed by Fowler and Beck has not been considered by the researchers. To date, several techniques have been proposed in the literature to identify opportunities for a limited set of refactoring activities. I am not aware of any reported discussion regarding why other activities might be ignored. Some of these activities, such as Rename Field, Rename Method, Inline Temp, and Add Parameter, are among the refactoring operations that have been found as the most applied refactoring activities in practice. These observations indicate a gap between the refactoring practice and the research in the area of identifying refactoring opportunities. I plan to integrate the existing code smell datasets and provide reliable code smell detection tools, which can be used to identify refactoring opportunities in search-based refactoring. This way the efficiency and correctness of search-based refactoring significantly increases.

Automated algorithm design and selection for SE tasks With the increasing number of proposed algorithms and methods for difficult software engineering tasks such as test data generation, fault localization, and search algorithm, selecting the best algorithm for a particular program instance is vital. A common approach to tackling the algorithm-selection problem is to treat it as a classification problem where each instance is described in terms of a vector of hand-designed features, and an instance's class indicates the best-performing algorithm. Algorithm selection successfully has been applied in many domains such as artificial intelligence and combinatorial problems. However, it has rarely been used for difficult SE tasks where there is no dominant algorithm for all problems' instances. The main challenges in this area are datasets of problem instances and solutions regarding the SE tasks and artifact representation for training appropriate algorithm selection models. I plan to create datasets and methods for automated algorithm selection in SE tasks such as test data generation. It is another way to amplify the performance of test data generators by running them only on SUTs that are suitable for the test data generation algorithm.

References

[1] M. Zakeri-Nasrabadi and S. Parsa, "Learning to predict software testability," in *2021 26th International Computer Conference, Computer Society of Iran (CSICC)*, IEEE, Mar. 2021, pp. 1–5, ISBN: 978-1-6654-1241-4. DOI: 10.1109/CSICC52343.2021.9420548. [Online]. Available: https://ieeexplore.ieee.org/document/9420548/.

[2] M. Zakeri-Nasrabadi and S. Parsa, "Learning to predict test effectiveness," *International Journal of Intelligent Systems*, 2021, ISSN: 0884-8173. DOI: 10.1002/int.22722. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/int.22722.

[3] M. Zakeri-Nasrabadi and S. Parsa, "An ensemble meta-estimator to predict source code testability," *Applied Soft Computing*, vol. 129, p. 109 562, Nov. 2022, ISSN: 15684946. DOI: 10.1016/j.asoc.2022.109562.

[4] M. Zakeri-Nasrabadi and S. Parsa, *CodART: Automated source code refactoring toolkit*, version 1.0.0, Jan. [Accessed: 2023-01-18]. DOI: https://doi.org/10.5281/zenodo.6508415. (visited on 01/18/2023).

[5] M. Zakeri-Nasrabadi, S. Parsa, and A. Kalaee, "Format-aware learn&fuzz: Deep test data generation for efficient fuzzing," *Neural Computing and Applications*, vol. 33, 5 2021, ISSN: 14333058. DOI: 10.1007/s00521-020-05039-7.

[6] S. Parsa, M. Zakeri-Nasrabadi, M. Ekhtiarzadeh, and M. Ramezani, "Method name recommendation based on source code metrics," *Journal of Computer Languages*, vol. 74, p. 101 177, Jan. 2023, ISSN: 25901184. DOI: 10.1016/j.cola.2022.101177.

[7] M. Zakeri-Nasrabadi and S. Parsa, *ADAFEST: A data-driven apparatus for estimating software testability*, version 1.0.0, Jun. [Accessed: 2023-01-18]. DOI: https://doi.org/10.5281/zenodo.6683950. [Online]. Available: https://m-zakeri.github.io/ADAFEST (visited on 01/18/2023).

[8] M. Zakeri-Nasrabadi and S. Parsa, *SENSA: Software entity name suggestion apparatus*, version 1.0.0, Nov. [Accessed: 2023-01-18]. [Online]. Available: https://github.com/m-zakeri/SENSA (visited on 01/18/2023).

[9] M. Shahidi, M. Ashtiani, and M. Zakeri-Nasrabadi, "An automated extract method refactoring approach to correct the long method code smell," *Journal of Systems and Software*, vol. 187, p. 111 221, 2022, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2022.111221. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121222000048.

[10] M. Zakeri-Nasrabadi, S. Parsa, E. Esmaili, and F. Palomba, "A systematic literature review on the code smells datasets and validation mechanisms," *ACM Computing Surveys*, May 2023, ISSN: 0360-0300. DOI: 10.1145/3596908.