Translating Quality-Driven Code Change Selection to an Instance of Multiple-Criteria Decision Making

Christos P. Lamprakos[a,*], Charalampos Marantos[a], Miltiadis Siavvas[b], Lazaros Papadopoulos[a], Angeliki-Agathi Tsintzira[c], Apostolos Ampatzoglou[c], Alexander Chatzigeorgiou[c], Dionysios Kehagias[b], Dimitrios Soudris[a]

[a] *School of Electrical and Computer Engineering, National Technical University of Athens, Greece*
[b] *Centre for Research and Technology Hellas, Thessaloniki, Greece*
[c] *Department of Applied Informatics, University of Macedonia, Greece*

Abstract

[**Context**] The definition and assessment of software quality have not converged to a single specification. Each team may formulate its own notion of quality and tools and methodologies for measuring it. Software quality can be improved via code changes, most often as part of a software maintenance loop. [**Objective**] This manuscript contributes towards providing decision support for code change selection given a) a set of preferences on a software product's qualities and b) a pool of heterogeneous code changes to select from. [**Method**] We formulate the problem as an instance of Multiple-Criteria Decision Making, for which we provide both an abstract flavor and a prototype implementation. Our prototype targets energy efficiency, technical debt and dependability. [**Results**] This prototype achieved inconsistent results, in the sense of not always recommending changes reflecting the decision maker's preferences. Encouraged from some positive cases and cognizant of our prototype's shortcomings, we propose directions for future research. [**Conclusion**] This paper should thus be viewed as an imperfect first step towards quality-driven, code change-centered decision support and, simultaneously, as a curious yet pragmatic enough gaze on the road ahead.

Keywords: Decision Support, Software Quality, Multiple-Criteria Decision Making

1. Introduction

Software engineering projects involve a diverse array of strategic decision making [8]. Opportunities appear all across the application development lifecycle. Particularly during the software maintenance stage, teams are often obliged to improve their product's quality.

"Software quality" refers to a software product's inherent characteristics. Several standards have been proposed, the most popular of which is ISO/IEC 25010:2011[1]. But quality definition does not equal quality measurement. Many approaches exist for selecting metrics relevant to each project's final formulation of product quality [1], also known as non-functional requirements (NFRs) [4]. In this paper we hold the assumption that the step following quality assessment is a set of code changes. If it is a fact that some code changes do harm NFRs like performance [3], then the converse should also be true; *it is possible to improve software quality via code changes.*

*Corresponding author
Email addresses: cplamprakos@microlab.ntua.gr (Christos P. Lamprakos), hmarantos@microlab.ntua.gr (Charalampos Marantos), siavvasm@iti.gr (Miltiadis Siavvas), lpapadop@microlab.ntua.gr (Lazaros Papadopoulos), angeliki.agathi.tsintzira@gmail.com (Angeliki-Agathi Tsintzira), ampatzoglou@uom.edu.gr (Apostolos Ampatzoglou), achat@uom.edu.gr (Alexander Chatzigeorgiou), diok@iti.gr (Dionysios Kehagias), dsoudris@microlab.ntua.gr (Dimitrios Soudris)

[1] https://www.iso.org/standard/35733.html

We treat the problem of deciding upon a pool of heterogeneous code changes (i.e. not all of them improving the same quality). Similar stances have been taken in recent work [7]. These approaches treat fully-automated, search-based software engineering while many teams prefer manual actions [6]. We propose formulating the task of **quality-driven code change selection** as an instance of Multiple-Criteria Decision Making [10]. We describe an abstract form of our methodology and then demonstrate a first prototype implementation addressing energy efficiency, technical debt and dependability. This prototype achieves inconsistent results; we qualitatively identify this behavior's root cause, and propose future work towards more reliable implementations.

Our main contributions are: (i) quality-driven code change selection is defined as a Multi-Criteria Decision Making (MCDM) problem and an abstract, extensible methodology built around the MCDM core is presented, (b) a prototype focusing on energy efficiency, technical debt and dependability, is described and evaluated, (c) related future work is proposed through a qualitative critique of the prototype's shortcomings.

2. Main Approach

The abstract form of our methodology is depicted in Figure 1. The following elemets are required:

- a set of NFRs[2]
- a set of code change[3] recommender processes, each targeting a separate non-functional requirement[4]
- a uniform set of impact models for quantifying the degree to which a code change could affect a quality attribute
- a set of functions that map candidate code changes to expected impacts on all quality attributes
- an MCDM algorithm

Each distinct quality attribute-centered recommender process generates a list of code change suggestions. Each candidate code change is given as input to all of the change impact functions, and thus estimates of how this change would affect each quality attribute are computed[5]. Thus we get a design space on which the trade-offs between each candidate change are imprinted. A more concrete example can be seen in Table 1, created in the context of the prototype that we present in Section 3.

This design space and the decision maker's (DM's) preferences are the inputs to the MCDM algorithm comprising our methodology's decision-making core.[6] The final output is a ranking of the code changes, based on each change's fitness to the DM's preferences, captured by the "Decision Value" field.

2.1. Decision-Making Core

We define two discrete spaces X and Ψ, comprising quality attributes and code changes respectively. We also define the set of real numbers $\mathbb{R}$ as the range of the change impact models. Each model is expressed by a function $f_{n \in X} : \Psi \rightarrow \mathbb{R}$. We denote each set of code change recommendations with $\Psi_i,\ i \in (1, 2, ..., N)$. There could exist empty sets or sets with overlapping elements but in any case, we are concerned with the union of all suggestion sets, which will be a subset of Ψ: $\Psi_p \coloneqq \bigcup_{i=1}^{N} \Psi_i$.

One can now visualize a table of $|\Psi_p|$ rows and $N + 1$ columns like Table 1[7]. This can be viewed as the *Design Space* block shown in Figure 1.

[2]From this point, we will use the term "non-functional requirements" for both the quality attributes' definition **and** the means for their assessment (metrics, thresholds, combination of metrics, etc).

[3]We define a code change as a **single**, contiguous edit in the source code. Thus, removing two unused variables from two non-neighboring spots of a program equals to two different code changes.

[4]Keeping a generic point of view, we do not impose any limitations on what these processes could look like. Maybe they are part of a static analysis tool, or an expert's opinion. The only invariant should be that each process generates a set of code

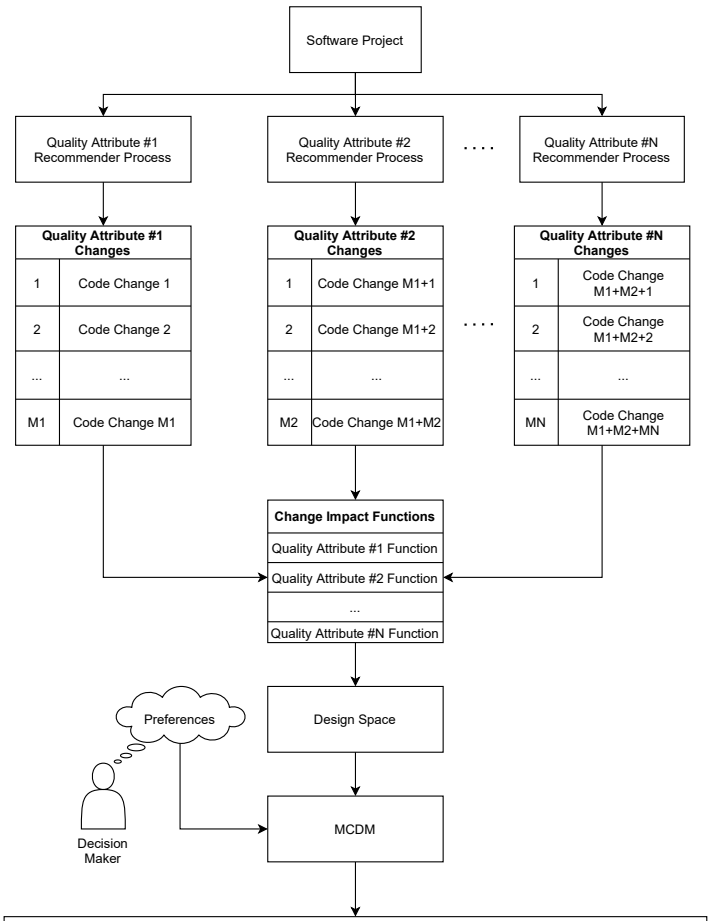

Figure 1: Functional diagram of the proposed method.

change suggestions.

[5]An alternative wording is the notion of "trade-off analysis".

[6]MCDM is related to whatever scenario requires decision support given a list of available options, and multiple criteria based on which the options are evaluated.

[7]This table was developed for our prototype instance. Things left abstract in Section 2 are here specified. For example, we treat refactoring operations instead of generic code changes. We pick fuzzy sets as the range of the change impact functions. Further details are provided in Section 3.

Table 1: A sample from the design space generated by this paper's prototype. Impacts on qualities were based on empirical measurements.

Code Change	**Energy**	**Technical Debt**	**Dependability**
Fix input/output issues	No Impact	Improve	Improve
Eliminate dead code segments	No Impact	Hinder	Improve
...	...	...	...

The value of each suggested change (that is, its fitness to the DM's preferences) can be expressed as:

$$\begin{aligned} V_j &\coloneqq w_1 \cdot f_1(j) + w_2 \cdot f_2(j) + ... + w_N \cdot f_N(j) \\ &\coloneqq \sum_{i=1}^{N} w_i \cdot f_i(j) \\ j &\in \Psi_p, \quad i \in X \end{aligned} \tag{1}$$

For Eq. 1 to be evaluated, the key component missing is the N-dimensional weight vector $\overrightarrow{w}$. This corresponds to a quantitative representation of the DM's preferences[8]. To compute $\overrightarrow{w}$, most MCDM algorithms require from the user to provide a form of her preferences with respect to the criteria under discussion. In our case, these criteria are the chosen quality attributes.

3. Prototype

This section describes an elementary implementation (prototype) of the abstract methodology presented above. Our prototype was developed in the context of the Horizon H2020 project SDK4ED—see Section 7. The specific software qualities chosen are energy efficiency (E), technical debt (TD) and dependability (D). It is not possible to elaborate on the details of our prototype's setup (metrics and derivation of each quality, change recommender processes, etc.). We do however provide a compact list of design decisions below[9]:

- ***NFRs:*** Energy consumption (milliJoules), principal technical debt (U.S. dollars), dependability (custom index). These were selected in the context of SDK4ED, the pilot use cases of which were either embedded, IoT or safety-critical applications. Technical debt aside (which could be considered universally important), the needs for high energy efficiency and optimal dependability are evident for such applications.
- ***Code change format:*** Refactoring. Architectural changes do not belong in this paper's scope[10].
- ***Change recommender process:*** Automatic identification of refactoring opportunities via static (technical debt, dependability) and dynamic (energy efficiency) analysis tools.
- ***Impact models:*** Lexicographical categories mapped to fuzzy sets following a similar approach to Shatnawi and Li [9]. As already mentioned, a sample of the design space may be seen in Table 1.
- ***Derivation of impact functions:*** Static empirical analysis via iterative application of refactoring operations on several open-source code segments, and subsequent quality assessment on the SDK4ED platform[11].

[8] Strange as it may sound, a central problem addressed by MCDM is precisely this quantification of preferences, which according to the literature is non-trivial if consistency and reliability are sought after.

[9] For further information, please feel welcome to consult the project's publicly available deliverable 6.4 at `https://sdk4ed.eu/documents/`.

[10] That said, we see no particular reason why the abstract version of our methodology could not be applied to architecture-oriented changes.

[11] See Footnote 9.

For the MCDM component, we implemented the Fuzzy Best-Worst Method (FBWM) algorithm by Guo and Zhao [5]. Like other MCDM algorithms, its inputs are an encoding of the DM's preferences and the impact-annotated design space of available options (in our case, refactoring opportunities). FBWM is flexible enough to support a wide spectrum of preference scenarios, ranging from single objective optimization (improve only one non-functional requirement) to joint, three-way optimization (improve all NFRs).

For the impact models' range, we defined three triangular membership functions in $[-1, 1]$ with equal overlaps of 0.5 units (*Hinder, No Impact, Improve*). For more details on FBWM (preference encoding, mathematical description), the reader may consult [5].

4. Results

We devised 3 'preference scenarios', each with a different ordering of importance on the qualities. The software project used for evaluation was Rodinia [2], a widely-used benchmark suite for heterogeneous computing. It comprises both CPU and GPU implementations of a wide array of computational kernels, from backpropagation to video editing. The data depicted below are **average** improvements measured after applying, for each benchmark, the proposed top-ranked refactoring[12].

Figure 2 displays the results retrieved from applying the top-ranked refactoring operations in three usage scenarios. *Usage Scenario* is a tuple showing the hierarchy of quality attribute importance as provided by the DM. From left to right, the elements denote a **best-worst-remaining** sequence. A result is positive if it respects the hierarchy posed by the DM, in the sense of improving the best NFR most and the remaining NFR less. As regards the worst NFR (in which the DM is the least interested), it is irrelevant whether it gets improved or not by the made decision. A result is negative if it does not respect the imposed preference hierarchy in any way.

According to Figure 2, our results are inconsistent. We provide brief comments for each scenario in a left-to-right fashion.

- **(TD,E,D)**: negative. The best criterion (TD) is indeed improved, but as regards the other two qualities, the hierarchy is reversed (worst gets improved, remaining gets hindered). The respective percentages also do not follow the hierarchy (TD should see the biggest improvement).
- **(E,D,TD)**: mostly negative, in a similar vein with the above. A hopeful detail is that percentages are more fitting here, since the best criterion (E) does indeed see the biggest improvement.
- **(D,TD,E)**: mostly positive. Best criterion (D) gets improved, worst criterion (TD) gets hindered. Of course, E should have seen smaller improvement than D. But having **both** of the top qualities improved, just in inverse proportions, is in our opinion the least painful of all possible headaches.

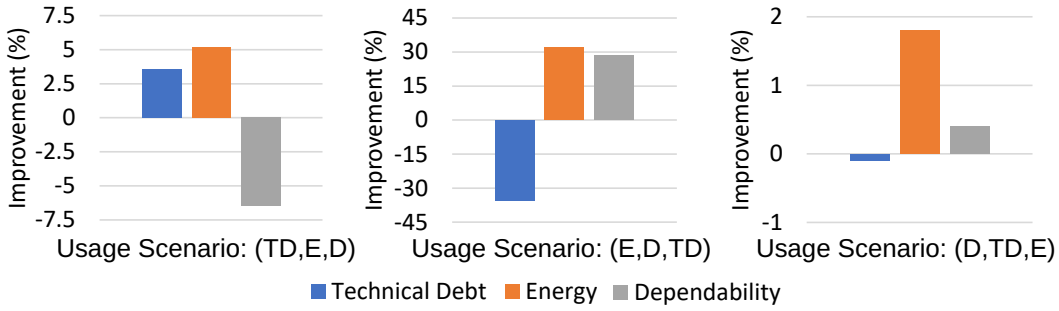

Figure 2: Experimental results: Average improvement (i.e. across all Rodinia benchmarks) of qualities after implementing the prototype's top-ranked suggestions.

[12]The SDK4ED platform provides analysis infrastructure for direct measurement of the stated software qualities. We performed measurements both before and after applying the MCDM-proposed code change.

5. Discussion and Future Work

Refactoring is known as a controversial means of quality improvement. We selected this code change paradigm due to our funding project's specifications. Future research could focus on alternative methods.

Moreover, a decision process is only as good as its inputs. We consider the main culprit behind our results' inconsistency the fact that our prototype uses a static, empirical model for the expected impact of refactoring on software quality. In this way it ignores the interdependencies in a specific project, which as a result distorts the design space (values in Table 1).

Change Impact Analysis (CIA) can be used for locating which parts of code would be affected by a candidate refactoring. The next step is to develop fine-grained local impact models. A far greater amount of information would then enrich the design space. This direction is orthogonal to the choice of the particular MCDM algorithm.

Last but not least, no comparison of our proposal with relevant works in the literature is presented. Acknowledging the issues discussed above does not leave any room for sparring with the state-of-the-art, even if similar tools exist (inconsistent results appear for the simplest of baselines).

6. Conclusion

This work concerns quality-driven decision making in the source code level during the maintenance and evolution stages of an application's development lifecycle. We presented an abstract methodology for integrating arbitrary software quality definitions in a workflow producing a ranked list of quality-enhancing code changes. The central component of our proposal is the multiple-criteria decision making theory and family of algorithms.

We also presented a prototype of our methodology targeting security, technical debt and energy efficiency. Even though our experimental results proved to be inconsistent, we believe our main idea to be worthy of further exploration.

7. Acknowledgements

This research was partially funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No 780572 SDK4ED (`www.sdk4ed.eu`).

References

[1] Basili, V. R. (1992). Software modeling and measurement: the goal/question/metric paradigm. Technical report.

[2] Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J. W., Lee, S.-H., and Skadron, K. (2009). Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE international symposium on workload characterization (IISWC)*, pages 44–54. Ieee.

[3] Chen, J. and Shang, W. (2017). An exploratory study of performance regression introducing code changes. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 341–352. IEEE.

[4] Chung, L. and do Prado Leite, J. C. S. (2009). On non-functional requirements in software engineering. In *Conceptual modeling: Foundations and applications*, pages 363–379. Springer.

[5] Guo, S. and Zhao, H. (2017). Fuzzy best-worst multi-criteria decision-making method and its applications. *Knowledge-Based Systems*, 121:23–31.

[6] Murphy-Hill, E., Parnin, C., and Black, A. P. (2011). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18.

[7] Ouni, A., Kessentini, M., Ó Cinnéide, M., Sahraoui, H., Deb, K., and Inoue, K. (2017). More: A multi-objective refactoring recommendation approach to introducing design patterns and fixing code smells. *Journal of Software: Evolution and Process*, 29(5):e1843.

[8] Ruhe, G. (2003). Software engineering decision support – a new paradigm for learning software organizations. In Henninger, S. and Maurer, F., editors, *Advances in Learning Software Organizations*, pages 104–113, Berlin, Heidelberg. Springer Berlin Heidelberg.

[9] Shatnawi, R. and Li, W. (2011). An empirical assessment of refactoring impact on software quality using a hierarchical quality model. *International Journal of Software Engineering and Its Applications*, 5(4):127–149.

[10] Triantaphyllou, E. (2000). Multi-criteria decision making methods. In *Multi-criteria decision making methods: A comparative study*, pages 5–21. Springer.