A Metrics-based Approach for Assessing Architecture-Implementation Mappings

J. Andres Diaz-Pace[1], Rodrigo Cian Berrios[2], Antonela Tommasel[1], Hernan C. Vazquez[2]

[1]ISISTAN Research Institute – UNCPBA University-CONICET
Campus Universitario Tandil – Tandil – Buenos Aires – Argentina

[2]Facultad de Ciencias Exactas – UNCPBA University
Campus Universitario Tandil – Tandil – Buenos Aires – Argentina

{andres.diazpace,antonela.tommasel}@isistan.unicen.edu.ar
cianrodrigo@gmail.com, hvazquez@exa.unicen.edu.ar

Abstract. *Several automated techniques for assisting engineers in creating mappings between source code (e.g., classes) and architecture elements (e.g., modules) have been proposed. They are generally assessed via precision and recall metrics. However, these metrics can only be evaluated post-mortem, i.e., once an expert created and validated all the mappings. In practice, given a set of mappings, engineers would like to assess their quality without (effortfully) validating the whole set. In this context, we explore a suite of quality metrics as an architectural fitness function for a ranking of mappings, which helps engineers select a useful list of those mappings. We empirically analyzed the evolution of our metrics in two projects using different mapping techniques.*

1. Introduction

The software architecture of a system is a blueprint that captures the main design decisions for satisfying the quality attributes (e.g., performance, modifiability, reliability, among others) posed by the system stakeholders [Bass et al. 2021]. These decisions are reflected in different architectural views (e.g., modules, components), which prescribe how the system should be implemented. Even when the architectural decisions are appropriately communicated to the development teams, there are usually *divergences* between the architecture "as designed" (e.g., the module view) and the architecture "as implemented" (i.e., the source code implementing those modules). When these divergences are not identified and properly controlled by the engineers, the benefits of architecture-centric development regarding the desired quality attributes tend to vanish, as the implementation no longer reflects the key design decisions. In this context, performing periodic conformance checks between the architecture and its implementation becomes necessary. A well-known approach for such checks is Reflexion Modeling (RM) [Murphy et al. 2001], which analyzes differences between the dependencies defined in the architecture blueprint and those found in the code. An RM prerequisite is the provision of mappings from source code elements to architecture elements.

Establishing the *architecture-implementation mappings* is often a tedious and error-prone task for engineers, as they must analyze many implementation details. Thus, several automated techniques (and tools) for assisting in creating and maintaining such

mappings have been developed [Olsson et al. 2022]. These techniques take different information sources (e.g., the system dependency graph, source code, or descriptions of architectural components) and have recently incorporated Machine Learning (ML) developments. Since the output of these tools is often a ranking of proposed mappings, their performance can be evaluated in terms of ML metrics, such as precision and recall, which require knowing the valid mappings for a system in advance (ground truth). Unfortunately, this situation seldom happens in a project, either because the ground truth is unknown or an expert must validate all the mappings proposed by a tool, which entails a considerable effort. Nonetheless, engineers would still need to assess the mappings suggested by a tool and select promising ones, without waiting until the end of the process (when all mappings are established and validated).

In this paper, we propose a suite of metrics for assessing the architecture-implementation mappings for a system, assuming they can be somehow ordered in a ranking. These metrics are taken from the literature [Guimarães et al. 2018, Lange and Chaudron 2004] and are agnostic of the particular technique or tool being used for mapping generation. The metrics include aspects such as: level of abstraction, consistency, completeness and code coverage. Unlike existing approaches, we do not require a ground truth of mappings. In our proposal, the metrics work as an *architectural fitness function* [Ford et al. 2017], providing hints about the quality (or usefulness) of the mappings. For instance, if a metric indicates that the number of architectural divergences increases as more mappings (of a given ranking) are accepted, the engineer might be prefer to keep only the top-k mappings of the ranking. Here, a related question is how many mappings from a ranking should be accepted. We approach this decision as a *parameter optimization problem*, which considers the evolution of the metrics and seeks to identify a cut-off point in the mapping ranking that (mostly) satisfies the engineer's preferences over the metric values. We evaluated the proposed framework in two projects using six mapping techniques. In those projects, the optimization procedure showed that $\approx 35\%$ of the mappings (of the rankings) can lead to satisfactory results, in terms of both useful mappings and precision/recall for the process. Although in initial stages, this work contributes to the practical application of architecture conformance techniques in projects and tools.

The rest of the paper is organized into 5 sections as follows. Section 2 provides the main concepts of architecture conformance and defines the mapping problem. In Section 3, our approach based on mapping metrics and optimization is presented. Section 4 reports on an initial evaluation of the approach with case-studies. Section 5 covers related work. Finally, Section 6 gives the conclusions and outlines future work.

2. Background

It is common to structure and organize systems in terms of views. On the one hand, we assume that the system implementation will be described by a *code view* that captures the main implementation units and how they depend on each other. An implementation unit can be materialized by a class (or a package), while a dependency is any method invocation or any other usage relation between classes (or packages). For instance, Figure 1 (right side) shows a simplified code view with five classes. On the other hand, we assume that the software architecture will be described by a *module view* that identifies intended system-level units, called *modules*, which group cohesive functions. These modules have

dependencies (or interactions) with each other, usually via provided/required interfaces. The example in Figure 1 (left side) shows a 3-tier architecture with three modules.

A standard approach for linking the architecture and code views is to structurally map them via the RM mechanisms. This approach requires engineers to establish one-to-many correspondences between code and architectural elements. In Figure 1, these mappings are exemplified by the dotted arrows from classes to modules. Based on the mappings, convergences and divergences between the relations prescribed by the architecture and those existing in the code can be discovered, as indicated by the green and red symbols, respectively. Divergences are also referred to as *architecture violations*.

We define below the main concepts for the architecture-implementation mapping problem, which serve as a conceptual framework for our proposal.

2.1. Formalization

Let $ArchView =< A, uses >$ be a graph (or tuple) that captures a static system view, in which $A = \{m_1, m_2, ..., m_N\}$ is a set of N modules and $uses$ is a binary relation between $m_i, m_j \in A$ denoting that the first module relies on services provided by the second module. Let $CodeView =< C, dependsOn >$ be a graph (or tuple) that represents a system implementation, in which $C = \{jc_1, jc_2, ..., jc_P\}$ is a set of P Java classes (with unique names) and $dependsOn$ is a relation between $jc_k, jc_l \in C$ indicating that the first class requires (or uses) methods defined in the second class. Dependencies between classes are often due to implementation needs.

The mapping can be seen as a matching between $ArchView$ and $CodeView$. More specifically, let $mappings = \{(jc, m) | jc \in C \wedge m \in A\}$ be a partial relation between elements at different levels of abstraction. This is a partial relation because some classes might have no correspondences with modules. A module, in turn, can relate to one or more classes. Along this line, the solution to the matching consists of finding (or approximating) $mappings$ as a function $f : C \rightarrow A$. Given the mappings, three possible types of differences between $ArchView$ and $CodeView$ arise: convergences, divergences, or absences. Divergences (violations) are of particular interest for architecture conformance because they imply that an intended module dependency is not followed in the code. Violations have quality-attribute effects concerning system modifiability, among other qualities [Brunet et al. 2012, Bass et al. 2021].

2.2. Mappings as similarity functions

RM does not prescribe how mappings should be established. For instance, engineers could rely on naming conventions of classes to link them to modules. Mapping generation is often done in a semi-automated way. In practice, either a tool suggests an initial set of mappings (or seeds) to the engineer, or the engineer provides an initial mapping that the tool then uses to infer additional mappings. From this perspective, the tool works as a *recommender system* [Robillard et al. 2014] that implements $f : C \rightarrow A$ and returns a *ranking of mappings*. We assume that each particular mapping has a goodness score in the range $[0..1]$, in which 1 refers to a perfect mapping (or similarity) between a pair of elements and 0 indicates a very unlikely mapping [Olsson et al. 2022].

A common strategy for generating likely mappings is to compute the similarity between all pairs (jc, m), being jc a class and m a module, respectively. In other words,

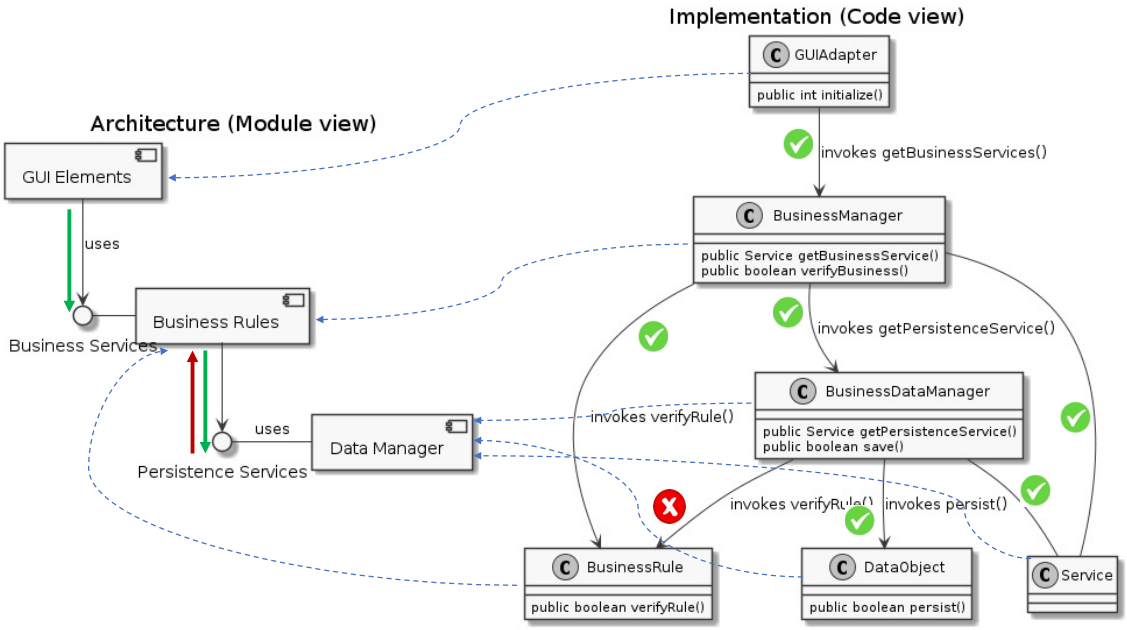

Figure 1. Example of mappings between architecture and implementation views.

we take the Cartesian product $C \times A$, under the constraint that any class can match only a single module. For a given class, the module with the highest similarity is chosen. Various similarity criteria between classes and modules can be applied. In this work, we focus on the following three similarity functions:

- **Linguistic similarity (L).** It compares the names of the class and module being mapped. To this end, the original names are split into tokens, and text similarity metrics are applied. In particular, we relied on a semantic metric provided by the SEMILAR toolkit [Rus et al. 2013].
- **Concern similarity (C).** A software concern is any aspect that impacts the design and implementation of a system, usually with cross-cutting effects on several artifacts [Robillard and Warr 2005]. For instance, a security concern might affect several classes and more than one module. Other examples of concerns include performance, persistence, or error handling. In terms of similarity, we say that a class matches a module if they "share" the same set of concerns.
- **Structural similarity (S).** It is based on the graph structure (or topology) of the module and code views. Similarity is treated as a graph matching problem. This function departs from an initial set of pairs of graph nodes (seeds) and then propagates the similarity to neighboring nodes, as proposed in [Melnik et al. 2002]. The seeds are either provided manually by the engineer or result from applying any of the previous functions.

As an example, Table 1 shows a matrix with the similarity scores computed by the three functions for the modules and classes of Figure 1. The best scores (for the mappings) are shown in bold. Note that the similarity functions might have different performance for the same pairs. This effect is caused by the heuristic nature of each function but it might also depend on class and module information. More details on the similarity functions above can be found in [Vázquez et al. 2015, Olsson et al. 2022].

	GUIAdapter			BusinessDataManager			BusinessRule		
	L	*C*	*S*	*L*	*C*	*S*	*L*	*C*	*S*
GUI Elements	**0.5**	**1.0**	**0.84**	0.25	0.0	0.52	0.28	0.0	0.0
Business Rules	0.0	0.0	0.0	**0.63**	**1.0**	**1.0**	**1.0**	**1.0**	**0.38**
Data Manager	0.0	0.0	0.0	0.71	0.0	0.0	0.28	0.0	0.37

Table 1. Examples of mapping scores using different similarity functions.

2.3. Performance evaluation

The traditional assessment of the mappings produced by a similarity function (as supported by a tool) is based on computing precision and recall metrics [Ricci et al. 2015], particularly when the ground truth (or reference mappings) is known or the mappings can be validated by an expert. The precision/recall formulas are given by Equations 1 and 2.

$$Precision = \frac{mappings \cap referenceMappings}{mappings} \tag{1}$$

$$Recall = \frac{mappings \cap referenceMappings}{referenceMappings} \tag{2}$$

In general, a tradeoff will exist between precision and recall, which is also affected by the goodness value of the mappings. High precision means that the suggested mappings are correct, while high recall implies that most of the real mappings were suggested. If high recall is accompanied by low precision, there will be some false positive mappings. In this regard, Figure 2 shows the evolution of these two metrics for a ranking of 120 class mappings (X axis) produced by a linguistic similarity function. Note also that the goodness value of the mappings (top-level chart) decreases, suggesting that the mappings beyond the 40th position (or with a goodness score < 0.5) should be considered spurious by the engineer. According to the charts, precision degrades significantly after that point ($k_{step} = 40 \approx 33\%$ of 120 mappings) at the cost of a slight improvement in recall. We refer to k_{step} as the *cut-off* point of the mapping ranking. Determining a good enough cut-off (for a given project) is important for the engineer to get the most of the performance of the mapping strategy (when the ground truth is not available), and it is the main driver of our approach.

3. Approach

Let us assume a scenario in which an engineer wants to map a code view to a module view using any automated strategy, and gets a ranking of suggested mappings. The engineer has to provide the architecture and code views as inputs for a mapping tool. Let us also assume that the tool is equipped with an arbitrary strategy for generating a ranking of mappings, as sketched in Figure 3. Since the real performance metrics (of the technique) will not be known in advance, the engineer can resort to "surrogates" for those metrics that we will refer to as a *fitness function* [Ford et al. 2017]. A fitness function is an instrument that provides an objective integrity assessment of some architectural characteristics. In our case, we are interested in conformance-related characteristics, and thus, we propose a *suite of metrics* for a ranking of mappings. Furthermore, these metrics can be computed at

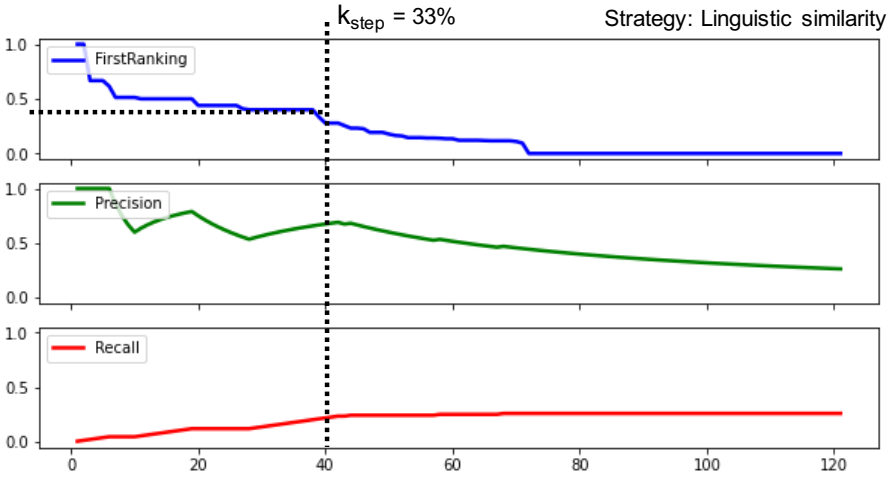

Figure 2. Evolution of goodness value of a ranking of mappings with respect to precision and recall.

different cut-off of the ranking to reflect how many mappings are accepted by the engineer and how much of the final metric values are retained. Note that the proposed metrics do not require having a ground truth, as it is the case of precision or recall.

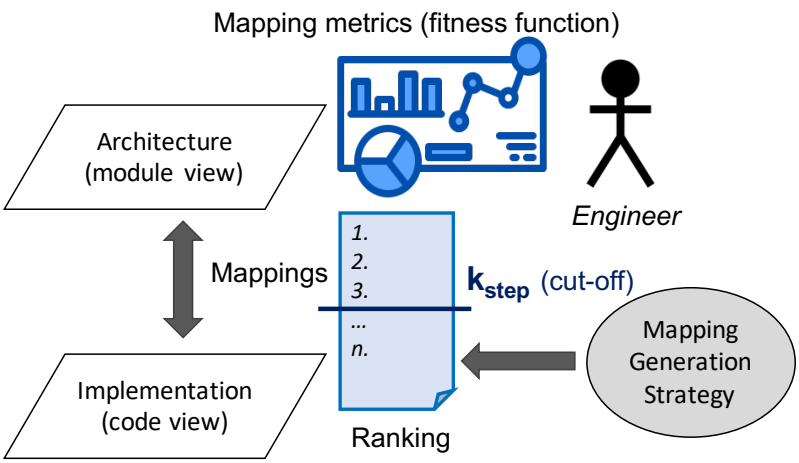

Figure 3. Main concepts of the proposed approach.

For instance, if we look at precision in Figure 4 for the case that the engineer would accept the full ranking (120 mappings), the global precision (of the technique) is 0.32 (*Precision*). Alternatively, we can assess the values of other metrics in the radar chart (e.g., *LevelAbstraction*, *Completeness*, *Consistency*, and *PackageCoverage*), as depicted by the blue area. A high value for *Consistency* suggests that the mappings introduced several violations, while a high value for *PackageCoverage* suggests that packages are evenly mapped to modules. If the engineer would instead take 17% of the mappings, then different metric values would be obtained, as depicted by the red area. Then, the engineer could either manually add the rest of the mappings, or re-run the tool with 17% of the mappings as seeds for getting more mapping suggestions. Specifically, when $k_{step} = 17\%$ we see that *Precision* goes up, *LevelAbstraction* stays the same, and *Completeness* and

Consistency go down. The mappings seem to have induced fewer violations while also lowering the package coverage. This means that considering a subset of the ranking can lead to better performance results. Although *Precision* is included for comparison purposes, its values will not be available, as the ground truth is unknown. In practice, the engineer should monitor the values of the four remaining metrics. Furthermore, an ideal k_{step} for the ranking should keep those metrics closer to their best reachable values.

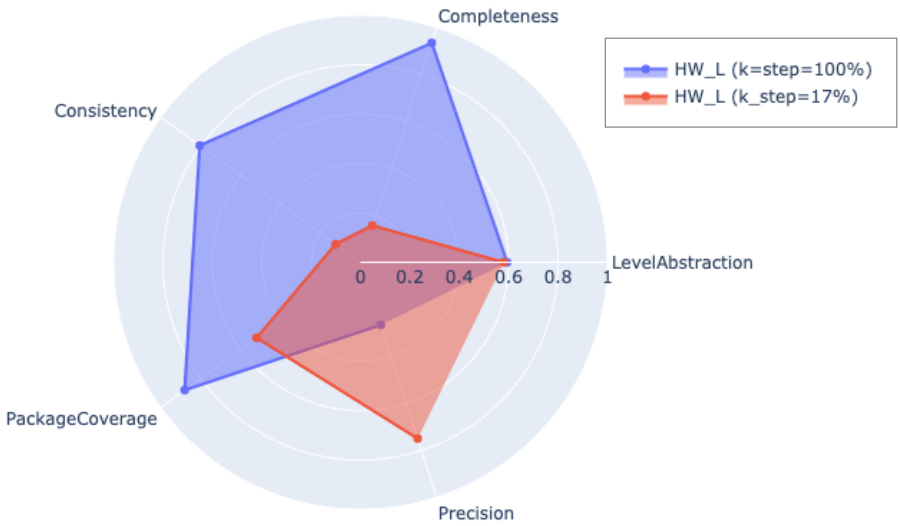

Figure 4. Comparison of metric values at two cut-off points of the ranking.

More formally, we define a fitness function $F = \{f_1, f_2, ..., f_m\}$ as a collection of metrics f_i, each one having the form $f_i\,(A, C, mappings_k) \rightarrow [0..1]$. The $mappings_k$ parameter results from applying a given mapping strategy S and then using k as the cut-off point for the corresponding ranking.

3.1. Suite of metrics

Our framework includes four metrics, in its current version, as described below. We chose these metrics inspired by [Lange and Chaudron 2004] and adapted code coverage ideas (from the testing field) to architecture conformance checking. All the metrics are monotonic with either increasing or decreasing trends.

- **Level of Abstraction**. This metric assesses the "distance" between the module and the code views, as computed by Equation 3. The smaller the value, the more detailed the abstraction level of the architecture, and vice versa. For instance, in the case of one single module mapped to most classes (which would not be very useful), $LevelAbstraction \approx 1$. In the opposite case, if there is a one-to-one mapping between each class and module, $LevelAbstraction = 0$. Ideally, an intermediate value for $LevelAbstraction$ is expected. This metric can be computed either for individual modules or the whole architecture. To this end, the code elements realizing each module ($\#classes(m_i)$) need to be identified.

$$LevelAbstraction(A) = \frac{1}{N}\sum_{i=1}^{N} LoA(m_i) = \frac{1}{N}\sum_{i=1}^{N}(1 - \frac{1}{\#classes(m_i)}) \qquad (3)$$

- **Completeness**. A given architecture is said to be complete if, for each module, there is at least a (mapped) code element that implements the module. The metric is computed as the ratio between the number of modules that are not mapped to any class ($\#unmapped(A)$) and the total number of modules, as indicated in Equation 4. $Completeness = 1$ means that all modules are mapped to at least one class. An architecture can be incomplete in two cases: i) some modules are yet to be implemented, and a partial implementation is being considered, or ii) a subset of the mappings has been accepted, and there are modules still unmapped.

$$Completeness(A) = 1 - \frac{\#unmapped(A)}{N} \tag{4}$$

- **Inconsistency.** An architecture is entirely consistent when no differences result from the mappings between the module and code views. Three types of differences are accounted for in this metric: i) dependencies between modules that are not present in the mapped classes (*absences*), ii) dependencies between classes that are not prescribed in the corresponding modules (*divergences*), and iii) dependencies between modules that also occur between the mapped classes but in the opposite direction (*divergences*). $Inconsistency$ is computed as the number of detected differences (for any of the above types), as shown in Equation 5. High values imply that the implementation fails to conform to the architecture blueprint (based on the mappings). This metric is normalized to $[0..1]$.

$$Inconsistency(A) = \#divergences(A) + \#absences(A) \tag{5}$$

- **Code coverage (**$PackageCoverage$**).** This metric refers to the number of code elements effectively mapped to at least one module ($\#unmapped(C)$). A code element can be a class or a group of classes, such as a package. In this work, we consider Java packages as proxies for "modules" in the code. $PackageCoverage$ bears similarities with the notion of test coverage, revealing how much code is mapped by the architecture. The metric is analogous to $Completeness$, but considering the implementation point of view, as shown in Equation 6. $PackageCoverage = 1$ indicates a full architectural coverage of the code.

$$PackageCoverage(C) = 1 - \frac{\#unmapped(C)}{P} \tag{6}$$

The metrics above can be easily implemented by tool (e.g., a plug-in for an IDE), which should store the architecture blueprint (e.g., as a list of modules and their relations) and keep track of their mappings to the source code. We are currently implementing a prototype Eclipse plug-in based on the ideas of the $ConcernMapper$ tool [Armentano et al. 2015, Robillard and Warr 2005]. The normalization parameters for the metrics can be determined either from the rankings or from the list of packages/classes recognized by the tool.

3.2. Optimal selection of the cut-off point

The analysis of the metrics when considering the order of the mappings of a given ranking shows that values increase (or decrease) with different trends. Depending on the system, certain metrics tend to reach their peak values earlier (in the ranking) while others move more slowly. In this context, the engineer should choose a cut-off point k_{step} that maximizes the chances for most metrics to reach their peak values. This choice is not straightforward, as the gain obtained for one metric might be in detriment of another metric, as shown in Figure 4. Let q_{metric} be the percentage of the metric final value that the engineer would like to achieve when selecting a subset of the mappings. For instance, if $PackageCoverage$ had a maximum value of 0.87 (when considering all the mappings), the engineer could prefer to get at least a 75% of that value when the cut-off is set. In general, the same kind of preferences can be stated for the four metrics in our framework.

The problem can be cast as a multi-objective optimization that depends on parameters k_{step}, q_{metric} and a particular mapping strategy S, given views A and C along with the $mappings$ between them. Furthermore, we can define upper, lower and target values as desirable for each metric to favor values closer to the target and penalize values outside the predetermined limits. This situation is exemplified in Figure 5 for the $PackageCoverage$ metric. The target is set to 75% of the (final) metric value (top-level chart), and two cut-offs are proposed, one above the target and another one below it. The "utility" of these points is quantified by means of a so-called *desirability function* [Derringer and Suich 1980] for $PackageCoverage$. A desirability function seeks to find preferred response values for another function (i.e., our metric). In the figure, the desirability function (bottom-level chart) favors metric values being greater than 0.6 ($q_{metric} \geq 75\%$). Different stereotypes for desirability functions have been proposed[1], such as response-maximization, response-minimization, or response-target-is-best, among others. We apply these stereotypes to our four metrics as follows: $LevelAbstraction$ is response-target-is-best, $Completeness$ is response-maximization, $Inconsistency$ is response-minimization, and $PackageCoverage$ is response-maximization.

Coming back to our fitness function $F = \{f_1, f_2, ..., f_m\}$, we translate each f_i to a desirability function counterpart d_i, and then combine them into a global function $D = \prod_i d_i$ to be maximized, as suggested in [Derringer and Suich 1980]. In this way, we formulate a single-objective optimization for D and perform a grid search of over different parameter configurations for k_{step} and q_{metric}. Since the rankings are dependent on the chosen mapping strategy S, it becomes a third parameter for the grid search.

4. Evaluation

To evaluate our proposed approach, we performed experiments with two Java systems, in which we executed different mapping strategies and also computed the suite of metrics described in Section 3.1 for the mappings resulting from those strategies. The main goal was to numerically determine k_{step} for each system while balancing the values of the metrics. We were also interested in analyzing q_{metric} and the role of the mapping strategy S with respect to the cut-off points.

[1] `shorturl.at/lmDG9`

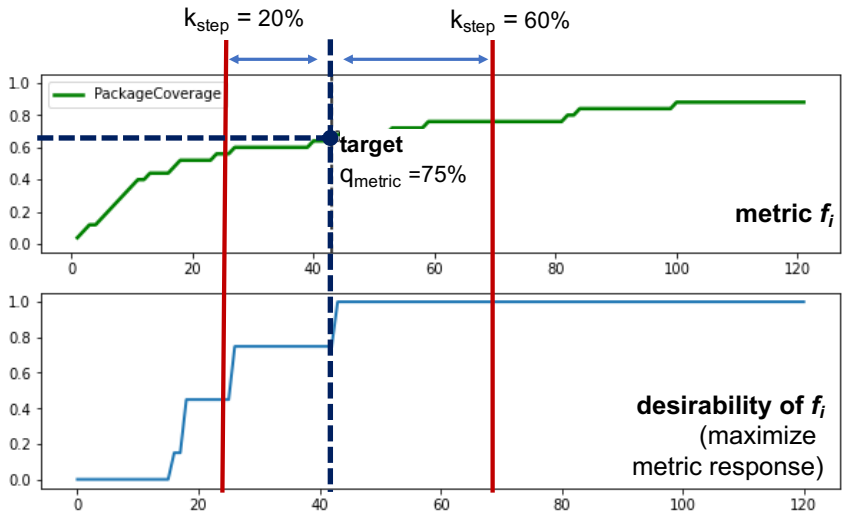

Figure 5. Quantifying preferred values for a metric using a desirability function.

The systems used as case-studies were Health Watcher (HW) and Mobile Media (MM) [Soares et al. 2002, Young 2005]. HW is a system for collecting and managing public health complaints and notifications, while MM is a software product line for mobile applications. The HW codebase has 120 classes (8697 LOC), and its architecture blueprint consists of 7 modules[2]. The MM codebase has 40 classes (3015 LOC), and its architecture blueprint consists of 15 modules[3]. Since the systems were relatively small (in size), we were able to compute different mapping strategies and understand the differences between one ranking or another. In both cases, reference mappings were available [Vázquez et al. 2015] in order to compute precision and recall metrics for comparison purposes. The reference mappings were determined by the original system developers, which is the standard practice in existing approaches to architecture conformance. We tested the strategies of Section 2.2, either in isolation or combined (e.g., applying linguistic and structural similarities in a sequence).

The optimization of the desirability functions was implemented in *Optuna*[4], a hyper-parameter optimization library for efficiently searching parameter configurations based on Bayesian techniques [Akiba et al. 2019]. Figure 6 depicts the values of the parameters explored by the optimization for both case-studies (darker points depict the progress of the Bayesian search). After 2000 trials, the best k_{step} was 35% and q_{metric} was 60% in both systems. This is the optimal cut-off for keeping 60% of all metric values, despite differences between HW and MM due to characteristics of their module and code views which might affect the performance of the similarity functions. It should be noticed that the mappings for the two systems revealed different levels of abstraction. We observed that a higher q_{metric} could be taken, at the risk of slightly decreasing the overall desirability function. This effect seems more noticeable in MM. Regarding the mapping strategy, a pure linguistic similarity (L) worked best in both systems. When comparing the strategies, Figure 6 shows that other linguistic variants combining structural and

[2]Notebook with experiments for HW: `shorturl.at/bdtG2`
[3]Notebook with experiments for MM: `shorturl.at/pDFI0`
[4]`https://optuna.org/`

hierarchical aspects (e.g., L_S_H) could work as well. This suggests that S can be an influential factor when selecting k_{step} in the rankings.

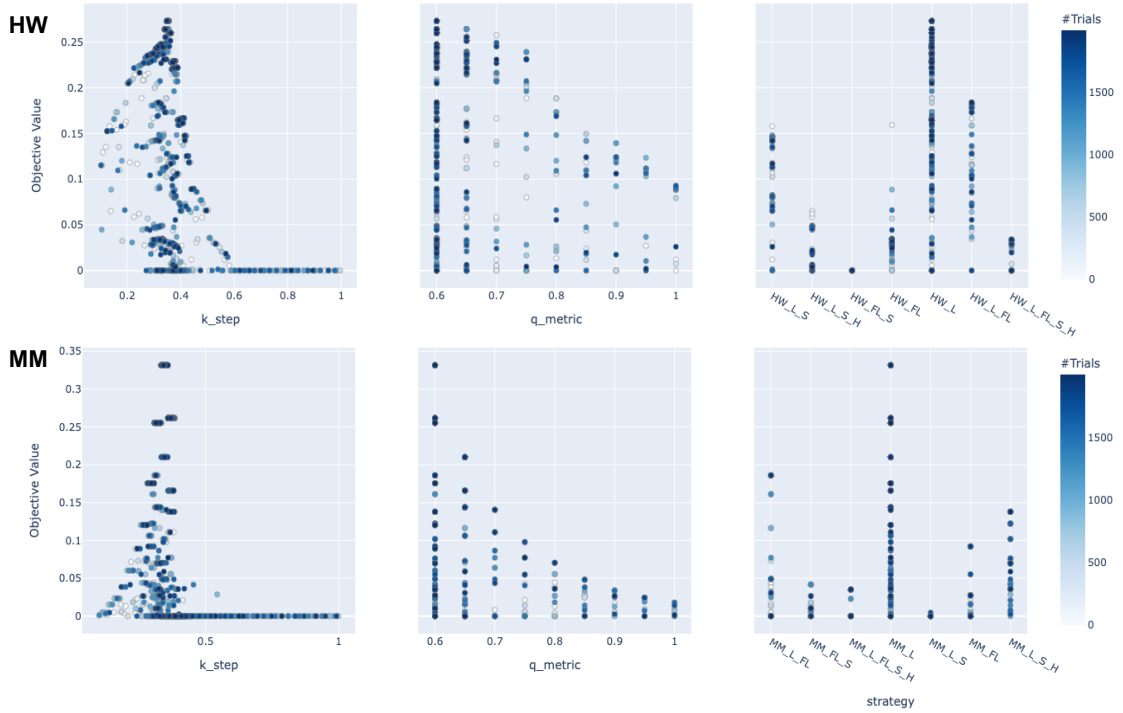

Figure 6. Slice plots with parameter values explored by *Optuna*.

In practice, our approach does not include precision and recall in its suite of metrics, and they were not part of the optimization process above. Nonetheless, we still used them in these experiments to have some kind of post-mortem performance indicators. Figure 7 shows the evolution of precision and recall for different mappings in the rankings. The X axis represents the ordered list of mappings (i.e., classes) considered in the rankings, which consequently affect the temporal evolution of the values. For instance, the top-left chart ($HW - precision$) shows that all mapping strategies degrade over time, but we can still distinguish two groups with different characteristics. The top-performing group starts with high precision values (when considering a few mappings) and goes down slowly around the 45th mapping position (which corresponds to 35% of the mappings). The strategies based on linguistic similarity (L) are included in this group. The other group reaches maximum precision values of 0.4 (which is deemed low) and degrades very quickly as more mappings are considered. These two groups were also detected for recall (top-right chart), although with the opposite effect, as recall increases steadily around the 45th position. Analogous trends can be observed for MM around the 15th position in the X axis.

In general, despite some variations between HW and MM, the plots show that most strategies yield a good precision for (approximately) the first half of the rankings, with some exceptions for a few low-performing strategies. We found correlations between the k_{step} determined by the optimization and the "natural" cut-offs exhibited by the plots of Figure 7. We should keep in mind that the computation of our fitness function is not based on precision nor recall. Thus, this result would indicate that the fitness function was

a good surrogate for the traditional performance metrics for architecture-implementation mappings. Nonetheless, more experiments and case-studies should be analyzed in order to confirm these trends.

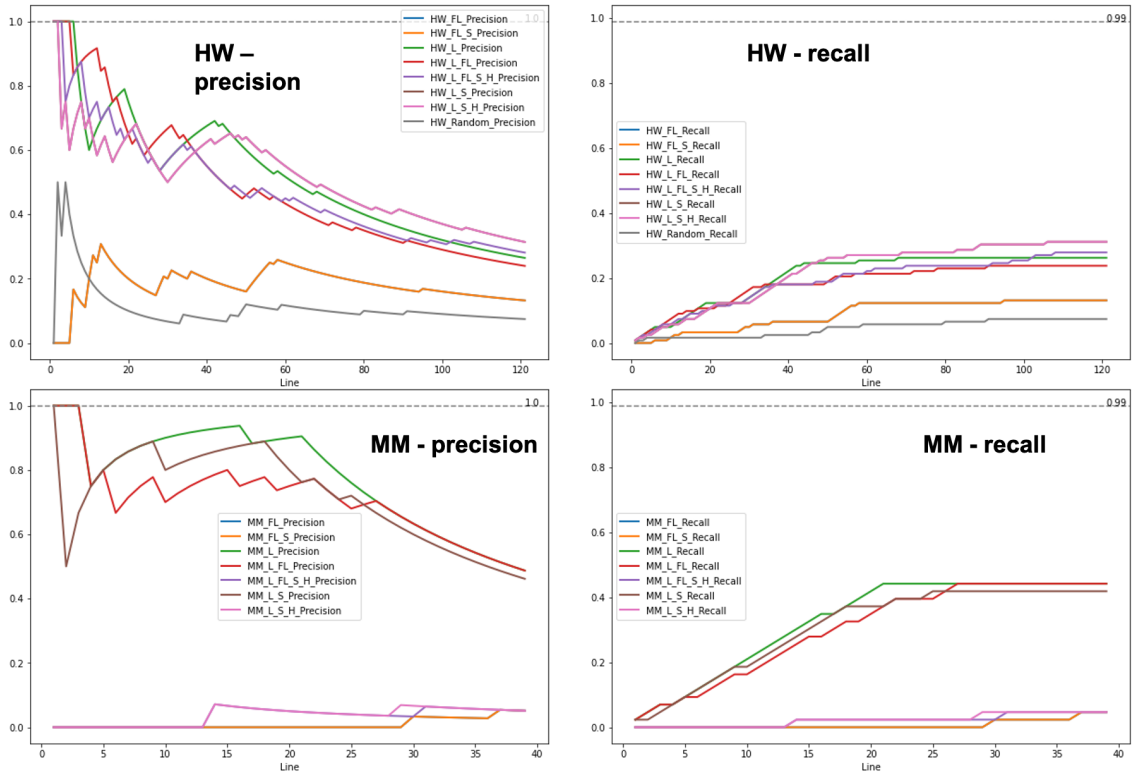

Figure 7. Precision/recall metrics of different mapping strategies for HW and MM.

5. Related work

Automated and semi-automated techniques for mapping code entities to modules have been proposed in the literature. These techniques traditionally rely on computing metrics and deriving a "similarity" or "module attraction" score for a code entity, and then assigning the entity to the module with the highest similarity. Other alternatives are based on training classification models. Various metrics have been introduced to this end. For example, [Christl et al. 2007] proposed to rely on the number of dependencies between entities and a modularity computation over the module dependency graph, while [Bittencourt et al. 2010] used latent semantic indexing (LSI). The main difference between existing metrics and strategies and our approach is that they rely on a mapping ground truth to assess performance. Existing strategies, however, are good candidates to extend our framework and the evaluation thereof. Furthermore, the notion of cut-off point for the mappings has not been explicitly addressed in the previous works.

Olson et al. [Olsson et al. 2022] extended both [Christl et al. 2007] and [Bittencourt et al. 2010] definitions to include module dependency information in their textual representation as part of an iterative mapping process. Although in [Olsson et al. 2022] the authors claim that considering the textual representation of modules allows to include semantic information in the orphan matching, the chosen LSI technique might not have been the best option, as it still requires a level of lexical

matching between texts, and sufficient documents for the statistical computations to be significant. In this sense, recent advances in software engineering embeddings (such as [Feng et al. 2020] or [Efstathiou et al. 2018]) could help to improve the semantic perspective of the analysis.

All the reported works in the literature rely on the existence of a ground truth to train and evaluate the corresponding models. As mentioned, this could require a considerable manual efforts from engineers. Conversely, in our proposal, we do not depend on having a ground truth, as our suite of metrics focuses on evaluating the quality of mappings. Thus, our framework intends to be more general and flexible in this regard.

6. Conclusions

In this paper, we have presented a framework based on metrics for assessing the quality of a ranking of architecture-implementation mappings, as generated by an automated strategy via similarity functions. This work aims at assisting engineers in selecting a good-enough list of mappings from the ranking rather than to develop alternative mapping strategies. We proposed four metrics based on characteristics of both the module and code views of the system. These metrics work as surrogates for precision and recall, assuming that the latter are not always available in real-life projects. Regardless of the mapping strategy being used, this is a practical contribution of our approach when compared to existing techniques.

We have formulated a multi-objective optimization of the metric values using desirability functions to determine the cut-off point for a ranking. We performed an initial evaluation with two relatively small Java systems that showed encouraging results. In the parameter optimization, both systems converged to similar values (35% of the mapping rankings for 60% of the metric values), which would indicate some regularity in the evolution of the metrics. As for the mapping strategy, using a linguistic similarity produced the best outcomes. Nonetheless, we observed different characteristics in the analyzed systems (e.g., levels of abstraction, number of modules and classes, or usage of naming conventions, among others). Therefore, these findings should be interpreted as preliminary and still require further experimentation.

As future work, we plan to add additional metrics as well as more advanced mapping strategies, such as [Olsson et al. 2022, Feng et al. 2020, Efstathiou et al. 2018], to the framework. These extensions could strengthen the approach and help us gain insights into how promising mappings should be chosen. Regarding empirical studies, we would like to investigate the effects of using the selected mappings as seeds for additional executions of the mapping strategy. Finally, to foster practical adoption of these techniques, the framework and metrics should be integrated with an architecture conformance tool (e.g., SonarQube, ArchUnit, or Codiga, among others).

References

Akiba, T., Sano, S., Yanase, T., Ohta, T., and Koyama, M. (2019). Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19, page 2623–2631, New York, NY, USA. Association for Computing Machinery.

Armentano, M., Soldavini, L., Díaz Pace, J. A., Vidal, S., and Marcos, C. A. (2015). Conformidad estructural de arquitecturas combinado con análisis de impacto de cambios. In *Simposio Argentino de Ingeniería de Software (ASSE 2015)-JAIIO 44 (Rosario, 2015)*.

Bass, L., Clements, P., and Kazman, R. (2021). *Software Architecture in Practice*. Addison-Wesley Professional, 4th edition.

Bittencourt, R. A., Santos, G. J. d., Guerrero, D. D. S., and Murphy, G. C. (2010). Improving automated mapping in reflexion models using information retrieval techniques. In *2010 17th Working Conference on Reverse Engineering*, pages 163–172.

Brunet, J., Bittencourt, R. A., Serey, D., and Figueiredo, J. (2012). On the evolutionary nature of architectural violations. In *2012 19th Working Conference on Reverse Engineering*, pages 257–266.

Christl, A., Koschke, R., and Storey, M.-A. (2007). Automated clustering to support the reflexion method. *Information and Software Technology*, 49(3):255–274. 12th Working Conference on Reverse Engineering.

Derringer, G. and Suich, R. (1980). Simultaneous optimization of several response variables. *Journal of Quality Technology*, 12(4):214–219.

Efstathiou, V., Chatzilenas, C., and Spinellis, D. (2018). Word embeddings for the software engineering domain. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 38–41, New York, NY, USA. Association for Computing Machinery.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. (2020). Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*.

Ford, N., Parsons, R., and Kua, P. (2017). *Building Evolutionary Architectures: Support Constant Change*. O'Reilly Media, Inc., 1st edition.

Guimarães, E. T., Vidal, S. A., Garcia, A. F., Pace, J. A. D., and Marcos, C. A. (2018). Exploring architecture blueprints for prioritizing critical code anomalies: Experiences and tool support. *Software: Practice and Experience*, 48:1077 – 1106.

Lange, C. and Chaudron, M. R. V. (2004). An empirical assessment of completeness in uml designs. *IET Conference Proceedings*, pages 111–119(8).

Melnik, S., Garcia-Molina, H., and Rahm, E. (2002). Similarity flooding: a versatile graph matching algorithm and its application to schema matching. *Proceedings 18th International Conference on Data Engineering*, pages 117–128.

Murphy, G., Notkin, D., and Sullivan, K. (2001). Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380.

Olsson, T., Ericsson, M., and Wingkvist, A. (2022). To automatically map source code entities to architectural modules with naive bayes. *Journal of Systems and Software*, 183:111095.

Ricci, F., Rokach, L., and Shapira, B. (2015). Recommender systems: introduction and challenges. In *Recommender systems handbook*, pages 1–34. Springer.

Robillard, M. P., Maalej, W., Walker, R. J., and Zimmermann, T. (2014). *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated.

Robillard, M. P. and Warr, F. W. (2005). Concernmapper: simple view-based separation of scattered concerns. In Storey, M. D., Burke, M. G., Cheng, L., and van der Hoek, A., editors, *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange, ETX 2005, San Diego, California, USA, October 16-17, 2005*, pages 65–69. ACM.

Rus, V., Lintean, M., Banjade, R., Niraula, N., and Stefanescu, D. (2013). SEMILAR: The semantic similarity toolkit. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 163–168, Sofia, Bulgaria. Association for Computational Linguistics.

Soares, S., Laureano, E., and Borba, P. (2002). Implementing distribution and persistence aspects with aspectj. *ACM Sigplan Notices*, 37(11):174–190.

Vázquez, H. C., Díaz Pace, J. A., and Marcos, C. A. (2015). Uso de ontologías para mapear una arquitectura de software con su implementación. *Jornadas Argentinas de Informática e Investigación Operativa*, 44.

Young, T. J. (2005). Using aspectj to build a software product line for mobile devices. In *University of British Columbia, Department of Computer Science*.