Multi-Dimensional Genetic Algorithm-Based Remodularization Integrating Structural, Semantic, and Behavioral Dependencies

Randeep Singh[1], Ganesh Khekare[2]

[1] Department of Computer Science Engineering, Lincoln University College, Selangor Darul Ehsan, Petaling Jaya, Malaysia;

[2] School of Computer Science and Engineering, Vellore Institute of Technology, Vellore, India

Email ID pdf.Randeep@lincoln.edu.my, khekare.123@gmail.com

Abstract: Architectural erosion in software systems is inevitable and it results in degraded modular quality that directly causes increased maintenance efforts along with reduced comprehensibility for the underlying software system. Software remodularization is a reverse engineering technique that helps in avoiding this issue. However, traditional software remodularization techniques primarily rely on structural dependencies and they generally neglect other important software features such as conceptual semantics and runtime behavior of a software system. This paper presents a **Genetic Algorithm (GA)-based multi-dimensional remodularization framework** that integrates structural, semantic, and runtime behavioral dependencies available in a software system. The framework evaluates seven dependency configurations arising from structural, semantic and behavioral dependency relations viz structural-only, semantic-only, behavioral-only, three pairwise combinations, and full integration. The evaluation of the remodularized solution is performed using three newly proposed quality metrics viz MQ+, SC, and CLA. Based on the carried out experimental evaluation, we demonstrate that full integration improves MQ+ by up to 25% over structural-only clustering while reducing cognitive load by nearly 40%.

Keywords—Remodularization, Genetic Algorithm, Software Architecture Recovery, Structural Dependency, Semantic Similarity, Behavioral Analysis.

Introduction

Over time, software systems may undergo architectural drift due to evolving requirements and incremental modifications that are generally made to incorporate changing requirements [1]. As a result, the original package structure degrades that unbalances coupling and cohesion quality characteristics. Automated remodularization techniques resolves this issue by aiming to restore degraded architectural quality by reorganizing classes into improved modules [2].

Most existing software remodularization approaches in literature rely primarily on **static structural relations** (e.g., call graphs, inheritance) as major dependency information extracted from source code [3]. Nevertheless, structural analysis alone ignores runtime interaction patterns and conceptual coherence. Recent studies suggest that integrating semantic information and behavioral dependencies [4] can significantly enhance quality of architectural recovery. Although these recent works provides several advances, the literature still lacks a systematic evaluation of (a) Individual dependency contributions, (b) Pairwise combinations, and (c) Full multi-dimensional synergy. Therefore, in this paper, we aim to address these research gaps by proposing a GA-based multi-dimensional remodularization framework. In this paper, we formulated the following research questions to answer:

RQ 1: To what degree does each considered dependency dimension individually contribute to remodularization quality and whether these contributions are consistent across systems?

RQ 2: Is there a synergistic effect when pairwise combinations of dependency are considered?

RQ 3: Does full multi-dimensional integration statistically and practically outperform all single-factor and pairwise configurations across all three evaluation metrics?

2. Related Work

In literature, there has been sufficient research on software remodularization and architecture recovery in the areas of static analysis, search-based optimization, semantic modeling, and dynamic analysis. Each of these research area are detailed as follows:

2.1 Structural Clustering Approaches

Structure dependency graphs are commonly used in early remodularization techniques to optimize cohesion and minimize coupling [5]. Evaluation of foundational cohesion or coupling was carried out by the Modularization Quality (MQ) metric [6]. Several improvements to this basic MQ quality metric have been proposed to address module size bias and edge weighting issues in the form of TurboMQ and extended MQ variants [7].

Graph partitioning and hierarchical clustering techniques are commonly applied to call graphs to recover underlying software structure [8]. Search-based methods such as hill-climbing and genetic algorithms have also been used to maximize modular quality of a software system [9]. However, structural-only approaches may produce clusters that ignore conceptual coherence especially in systems with cross-cutting concerns [10].

2.2 Semantic-Based Modularization

The researchers introduced semantic analysis based on textual similarity as a means of addressing the limitations of purely structural clustering [11]. Latent Semantic Indexing (LSI) and topic modeling have been applied to source code of a software system in order to detect underlying conceptual groupings of software elements (classes) [12, 13].

In recent years, code embeddings and transformer-based models have been used to capture deeper semantic representations [14, 15]. Semantic-based approaches seek to optimize lexical similarity without enforcing architectural constraints, which can result in modules that lack structural cohesiveness and low semantic cohesion.

2.3 Behavioral and Dynamic Dependency Integration

Using runtime behavior allows us to capture execution-based relationships, such as reflection and event-driven callbacks that static analysis might miss for a software [16]. A reverse engineering method based on trace-based clustering has been proposed to improve architectural recovery through dynamic call graphs for a software system [17]. The generalizability of behavioral dependencies depends strongly on the test coverage and workload representativeness [35].

2.4 Search-Based Multi-Objective Remodularization

A search-based software engineering (SBSE) technique uses evolutionary algorithms to explore search spaces to carry out software remodularization [18]. In order to optimize both structural metrics and semantic cohesion simultaneously during software remodularization, genetic algorithms like NSGA-II have been used [19].

Despite this, most multi-objective techniques still highlight structural optimization and infrequently consider runtime behavioral evidence in unified frameworks [12, 19]. Further, there is a lack of

comprehensive ablation studies that evaluate the differences between individual and combined dependency dimensions [19].

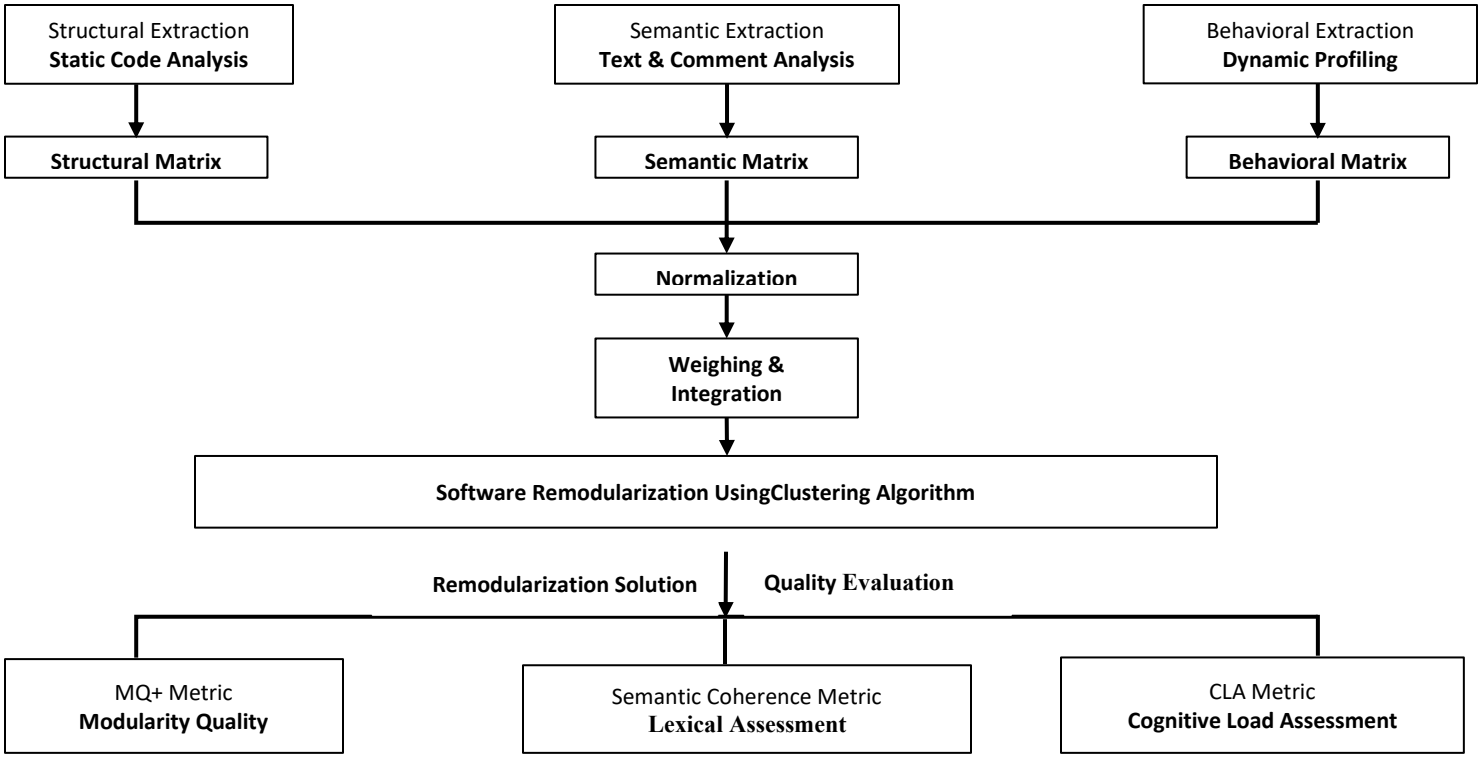

Figure 1. Proposed GA-based multi-dimensional software remodularization pipeline.

3. Proposed Methodology

Our methodology as depicted in Figure 1 consists of four main stages: **(1) Dependency Extraction**, **(2) Normalization**, **(3) GA-based Integration & Clustering**, and **(4) Quality Evaluation**. We first parse each system to extract structural dependencies, semantic affinities, and behavioral traces. Each of these dependencies forms a weighted graph on classes, and the edges denote similarity or dependency strength. These graphs are then normalized to a common scale. Depending on the configuration, we integrate one or more graphs into a composite similarity network, and performed GA-based clustering. Finally, we evaluate the clusters using three newly proposed quality metrics. The working of each of these four main stages is explained as follows:

3.1. Dependencies Extraction

For each of the class pairs (C_i, C_j), we extracted three types of dependency information. **Structural dependencies** $S_{struct}(C_i, C_j)$ are derived through static analysis of the source code. It covers identifying method calls, inheritance relationships, interfaces, field accesses, parameter dependencies, and return types information from underlying source code. The interaction between classes is modelled as a weighted structural dependency graph, where nodes represent classes and edges represent their coupling strength between corresponding pair of classes.

Source code components are analyzed lexically to determine their semantic dependencies. Lexical analysis involves tokenizing class names, method names, variable identifiers, and comments. Next, the tokens are preprocessed (stop-words are removed and normalized) and represented using the term-frequency inverse-document-frequency (TF-IDF) matrix. Finally, cosine similarity values between class vectors are

computed in order to obtain semantic similarity values $S_{sem}(C_i, C_j)$. While structural dependencies can clearly be seen, semantic similarity can reveal hidden relationships that might not be obvious from direct code interactions

Behavioral dependencies in this paper are derived from dynamic execution profiling of a software system. During dynamic profiling, the system is instrumented and run using existing test suites or typical runtime scenarios. Finally, execution traces are gathered to document method calls and patterns of class co-activation for a given software system. In this paper, we represented this information in the form of a behavioral dependency matrix $S_{behav}(C_i, C_j)$. This dimension highlights runtime coupling that may not be evident from static information available in source code like interactions driven by events or callbacks at the framework level.

3.2. Normalization

To ensure comparable results, a normalization phase is carried out by us in this paper because these three dependency matrices originated from heterogeneous sources and operate at different scales. Min-max normalization is used to transform each similarity matrix so that values lie within the range of [0, 1]. In addition, we also filtered noise in this paper by retaining only significant dependency relationships after thresholding weak values after normalization (tunable parameter).

3.3. GA-based Integration and Clustering

After normalization, the select dependency matrices are integrated to generate a composite similarity graph. For a given configuration, the composite similarity at the class level is computed as a weighted linear combination of the normalized structural, semantic, and behavioral similarities. The integration weights for different similarities are determined by the configuration being evaluated. As an example, only the structural matrix contributes to the composite graph in the structural-only configuration, whereas all three dimensions are equally weighed in the full integration configuration. In order to identify optimal module partitions, the proposed evaluation framework uses a Genetic Algorithm. During implementing GA, we encoded candidate solutions as chromosomes that represent classes assigned to different software modules as integer string. The fitness of every chromosome is determined through a composite objective function that balances three factors. It simultaneously maximizes structural modular quality, enhances semantic coherence, and reduces cognitive load.

3.4. Quality Evaluation

This phase analyzes the quality of the remodularized structure created by the Genetic Algorithm through clustering by using three new quality metrics.

Modularization Quality (MQ+) Metric

MQ+ is an enhanced version of the classical Modularization Quality metric used in architecture recovery research. It evaluates the balance between intra-module cohesion and inter-module coupling. The overall system-level MQ+ metric for a remodularized solution having k modules is mathematically defined as follows:

$$MQ+=\frac{1}{k}\sum_{p=1}^{k}\frac{Intra(M_p)}{Intra(M_p)+Inter(M_p)}$$

MQ+ ranges between 0 and 1. Higher values indicate stronger cohesion within modules and weaker coupling across modules.

Semantic Coherence (SC) Metric

Semantic Coherence evaluates the conceptual consistency of classes grouped within the same module. In the proposed framework, semantic similarity between classes is computed using cosine similarity over TF-IDF representations of source code text.

Cognitive Load Assessment (CLA)

It measures the understandability of the modular structure. It is based on the premise that modules with excessive external dependencies or imbalanced size increase developer mental effort. For module M_p, the module size complexity (MSC) is defined as the ratio of modular size to the total number of classes in the software system. Similarly, the external dependency ratio (EDR) is mathematically expressed as:

$$EDR(M_p) = \frac{Inter(M_p)}{Intra(M_p) + Inter(M_p)}$$

Using these two values, the system-level CLA is mathematically expressed as follows:

$$CLA = \frac{1}{k}\sum_{p=1}^{k}\left(EDR(M_p) + MSC(M_p)\right)$$

4. Experimental Setup

This section describes the subject systems, dependency extraction process, experimental configuration, and execution protocol used to evaluate the proposed multi-dimensional GA-based remodularization framework.

4.1. Experimental Dataset

To ensure generalizability and external validity, three widely used open-source Java systems are selected in this paper for experimentally validating the proposed approach, and Table 1 depicts the necessary details about these subject systems.

Table 1: Considered subject systems.

System	Version	Domain	LOC	#Classes	#Packages	Test Coverage	#Structural Edges	#Unique Terms	#Dynamic Edges
JUnit	4.12	Unit Testing Framework	10,482	187	12	88%	1,842	3,214	1,129
Apache Commons IO	2.6	I/O Utility Library	24,936	410	26	81%	5,476	5,873	2,784
ArgoUML	0.34	UML Modeling Tool	52,318	785	39	63%	16,932	12,486	9,457

4.2. Genetic Algorithm Configuration

The Genetic Algorithm was implemented in Java and executed under identical parameter settings across all configurations to ensure fairness. The population size is taken as 100, and the number of generations is considered as 200. The crossover and mutation rate parameters are assigned values of 0.80 and 0.05, respectively. Tournament size is considered as 3, and elitism parameter is preserved as the top 5% per generation. Further, each experiment was executed five times independently to account for stochastic variation.

4.3. Experimental Protocol

For each subject system, seven configurations are evaluated, and for each configuration, we used identical GA settings and termination criteria in this paper. The only difference across experiments was the similarity matrix integration weights. The number of modules was initialized equal to the original package count to maintain comparability with the baseline architecture. For each configuration, we reported the

mean value across five runs along with the standard deviation recorded, and performed paired t-tests against the structural-only baseline.

5. Results and Interpretations

This section provides a comprehensive quantitative analysis of the seven considered dependency configurations. Standard deviation among obtained result values are also observed and presented here in order to demonstrate stability of GA for different configurations. The results obtained are presented in a system-wise manner, utilizing MQ+, semantic coherence, and the CLA metric to assess the experimental outcomes. Tables 2 in this paper illustrate the experimental findings for different quality metrics (as JUnit/Commons-IO/ArgoUML) along with standard deviation value.

Table 2: Comparative experimental results across considered systems.

Configuration	MQ+ (↑)	SC (↑)	CLA (↓)
Structural-only	0.66±.012/0.65±.013/0.65±.013	0.45±.015/0.43±.017/0.43±.017	0.81±.018/0.80±.020/0.80±.020
Semantic-only	0.58±.011/0.59±.012/0.59±.012	0.76±.014/0.78±.016/0.78±.016	0.75±.016/0.73±.018/0.73±.018
Behavioral-only	0.53±.013/0.55±.014/0.55±.014	0.49±.013/0.50±.014/0.50±.014	0.66±.020/0.64±.021/0.64±.021
Structural + Semantic	0.73±.010/0.74±.011/0.74±.011	0.83±.012/0.84±.013/0.84±.013	0.69±.017/0.67±.016/0.67±.016
Structural + Behavioral	0.70±.011/0.71±.012/0.71±.012	0.61±.013/0.62±.015/0.62±.015	0.57±.015/0.56±.017/0.56±.017
Semantic + Behavioral	0.64±.012/0.66±.013/0.66±.013	0.87±.011/0.88±.012/0.88±.012	0.61±.014/0.60±.015/0.60±.015
All Three Combined	**0.81±.009/0.82±.010/0.82±.010**	**0.91±.010/0.92±.011/0.92±.011**	**0.49±.012/0.48±.013/0.48±.013**

The experimental findings consistently indicate that the integration of multi-dimensional dependencies greatly enhances the quality of remodularization across all subject systems. The structural-only configuration yields reasonable MQ+ values; however, it suffers from low semantic coherence and a high cognitive load. Furthermore, the semantic-only configuration optimizes conceptual grouping but compromises structural integrity, leading to moderate MQ+ and relatively high CLA. In the behavioral-only configuration, classes that co-execute frequently reduce cognitive load, yet MQ+ is the lowest. The "Structural + Semantic" combination improves both cohesion and conceptual consistency, while the "Structural + Behavioral" combination significantly decreases cognitive load. Hence, overall we can conclude that the comprehensive multi-dimensional integration consistently achieves the highest MQ+, the greatest semantic coherence, and the lowest cognitive load across all systems. On average, MQ+ shows an improvement of approximately 25%, while CLA decreases by nearly 39% in comparison to the structural-only baseline. These findings suggest that structural, semantic, and behavioral dependencies are complementary rather than redundant.

6. Conclusion & Future Works

In this paper we proposed a multi-dimensional remodularization framework that integrates structural, semantic, and runtime behavioral dependencies. Experimental evaluations reveal that the comprehensive integration of structural, semantic, and behavioral information consistently surpasses single and pairwise configurations across all evaluation metrics. The all dependency integration configuration achieved an approximate 25% enhancement in MQ+ and nearly a 39% decrease in cognitive load results. These findings validate that architectural quality is fundamentally multi-dimensional in nature and require systematic integration of dependencies. Further, their integration within a GA-based global search framework yields modular decompositions that are structurally robust, conceptually significant, and cognitively manageable.

Several potential directions remain for extending this research. Firstly, adaptive weight learning could be implemented to automatically ascertain the relative contributions of different dependencies. Secondly, a multi-objective evolutionary algorithm might be utilized to independently optimize multiple trade-offs among architectural objectives. Lastly, more sophisticated semantic representations could enhance the quality of conceptual clustering.

REFERENCES

1. G. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner, "Bunch: A clustering tool for the recovery and maintenance of software system structures," in *Proc. IEEE ICSM*, 1999, pp. 50–59.
2. S. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Proc. WCRE*, 2000, pp. 258–267.
3. Nandini, A., Singh, R., & Rathee, A. (2024). Code smells and refactoring: a tertiary systematic literature review. *International Journal of System of Systems Engineering*, *14*(1), 83-143.
4. G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Using structural and semantic measures to improve software modularization," *Empirical Software Engineering*, vol. 18, no. 5, pp. 901–932, 2013.
5. A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proc. ASE*, 2001, pp. 107–114.
6. N. Tsantalis and A. Chatzigeorgiou, "Identification of extract method refactoring opportunities," in *Proc. CSMR*, 2009, pp. 119–128.
7. Z. Wang et al., "CodeBERT: A pre-trained model for programming and natural languages," in *Proc. EMNLP*, 2020, pp. 1536–1547.
8. C. Patel, A. Hamou-Lhadj, and J. Rilling, "Software clustering using dynamic analysis and static dependencies," in *Proc. ICSM*, 2009, pp. 000–000.
9. Y. Xiao, J. Sun, and J. Zhao, "Dynamic analysis-based software clustering," *Journal of Systems and Software*, vol. 85, no. 4, pp. 761–775, 2012.
10. M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 1–61, 2012.
11. J. A. Dallal, "A systematic review of fault prediction studies," *Empirical Software Engineering*, vol. 17, no. 4–5, pp. 375–430, 2012.
12. Y. Kessentini et al., "Search-based refactoring using recorded code changes," *Journal of Systems and Software*, vol. 125, pp. 215–231, 2017.
13. L. Aversano et al., "Clustering support for software architecture recovery," *Journal of Systems and Software*, vol. 83, no. 10, pp. 1721–1734, 2010.
14. Y. Liu et al., "GraphCodeBERT: Pre-training code representations with data flow," in *Proc. ICLR*, 2021.
15. Z. Chen et al., "Evaluating deep learning for source code understanding," *IEEE Software*, vol. 39, no. 3, pp. 98–105, 2022.
16. N. Anquetil et al., "Static vs dynamic analysis for architecture recovery," in *Proc. ICSME*, 2016.
17. R. Oliveto et al., "Traceability link recovery using IR methods," *Empirical Software Engineering*, vol. 15, no. 5, pp. 565–610, 2010.
18. R. Abreu and R. Premraj, "Search-based refactoring strategies," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 789–804, 2011.
19. Y. Kessentini, M. Wimmer, and J. Rilling, "A multi-objective approach for software remodularization," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1609–1646, 2015.