

# Towards Probabilistic Assessment of Modularity

Kevin Hoffman   Patrick Eugster

Purdue University, Department of Computer Science  
305 N. University St., West Lafayette, IN 47907  
kjhoffma@cs.purdue.edu   peugster@cs.purdue.edu

## 1. Introduction

With recent trends showing increasing rates of software evolution and complexity, modularity is more important than it ever has been for on-time, on-budget software development. Assessing modularity is important for both evaluating current software quality and planning future changes. This latter use case is becoming more relevant as open-source models become more common and developers contribute to code with which they may be quite unfamiliar.

Software engineers tasked with implementing specific changes to unfamiliar software might ask questions such as:

- Q1** Which program elements are the most depended upon across the system? ... within a component?
- Q2** Which program elements are the most fragile (likely to change)?
- Q3** Which program elements are most likely to be affected by ripple effects given a set of changes?
- Q4** Which program elements were most influential during some execution of a test case?

Traditional modularity metrics do not provide immediate answers to these questions. To assist in answering such questions, we propose new modularity metrics based on probabilistic computations over weighted design structure matrices (DSMs) [1], also known as dependency structure matrices. We define how these metrics are calculated, discuss some practical applications, and conclude with future work.

## 2. Probabilistic Analysis of Modularity

DSMs are an established technique for assessing the modularity of a software system [7, 8]. DSMs model the pairwise dependencies between design elements and program elements in matrix format. Both rows and columns are labeled with these elements. Each matrix cell models the degree to which the element corresponding to that row depends on the element corresponding to that column. Clustering and partitioning algorithms can be applied to organize dependencies within the system. Tools such as LDM [7] generate DSMs by examining static dependencies within source code. In LDM dependency weights can be counts of individual dependencies (e.g., number of method calls, class instantiations, etc.)

or can be knowledge based (such as the number of classes referenced).

Our goal is to extrapolate from these individual pairwise dependencies more information about the dependent structure of the system as a whole. We propose that the DSM can be used to form a probabilistic model measuring system-wide dependency, termed the *Dependency Propagation Ranking*, as follows: Define the *Assessment Graph* as a weighted directed graph in which the vertices are design or program elements and the edges represent the probability of the source element affecting the target element if the source element changes.

These edges are derived from the dependency information in the DSM, such that if element  $A$  depends on element  $B$  in the DSM, then an edge will be created from vertex  $A$  to vertex  $B$  in the Assessment Graph. The weight of the edge is a function of both the corresponding weight in the cell of the DSM and a normalization algorithm. Note that there never is more than one edge between any given pair of vertices. This allows the graph to be represented as a *weighted edge matrix*, where the value of the cell at row  $i$  and column  $j$  is the weight of the edge from  $i$  to  $j$  or 0 if there is no such edge.

The weighted edge matrix,  $M$ , of the Assessment Graph can represent the stochastic matrix of a time-homogeneous Markov chain if the following two conditions are met:

1. Normalized edge weights: For each vertex  $i$ , the sum of all of the probabilities moving from vertex  $i$  to other vertices must be exactly 1. In other words, the values must sum to 1 for each row in the matrix. This is accomplished by applying a normalization algorithm, with the simplest one being  $M'_{i,j} = M_{i,j} / \sum_k M_{i,k}$ .
2. All vertices must have at least one edge: In our graph construction algorithm above, elements that did not depend on any other elements would have zero edges in the graph and thus be sinks. This can be fixed by adding edges from each sink vertex to every other vertex in the graph, giving weight  $1/|M|$  to each edge. This does not affect the metric values produced from the probabilistic model [2].

This Markov chain now represents the probabilistic propagation of dependencies within the system. A full armament of probabilistic analyses can be applied to the Markov chain.

One such analysis is the calculation of the stationary distribution of the Markov chain, which would provide a global ranking of dependency (such that components that are more highly depended upon are ranked higher). This is analogous to the representation and calculation of global web page ranking in PageRank [6] and of global trust values in EigenTrust [4]. Computational calculation is straightforward and proceeds by iteratively calculating the principle left eigenvector of  $M$ . Care must be taken to ensure that the calculation will converge. Applying a *damping factor* in the iterative calculation ensures convergence, and we postulate that it will not unduly affect the usefulness of the results, as was the case in [6, 4]. Dependency Propagation Ranking can help answer motivating question Q1.

The same algorithm can be used to calculate other probabilistic metrics by (a) varying how weights in the input DSM are generated, (b) considering only certain types of dependencies when generating the DSM, (c) varying what vertices in the Assessment Graph represent, (d) changing how Assessment Graph edges are constructed from the DSM, and (e) adjusting how weights are normalized. By inverting edge direction in the Assessment Graph, we can model the probabilistic propagation of changes in the system and rank those components most likely to be affected by changes, termed Impact Propagation Ranking, helping with Q2.

Also, we define the *Impact Shift Ranking* as the difference between the Impact Propagation Ranking values for two versions of a program, allowing developers to immediately understand the relative impact on ripple affects for a set of changes (Q3). Finding a way to 'un-normalize' the Impact Propagation Ranking values would allow us to define Aggregate Impact as the total of all un-normalized Impact Propagation Ranking values, as a measure of total system susceptibility to ripple effects.

Information from static analyses, such as a data flow analysis, could be used to create weighted DSMs, allowing for measurement of the propagation of indirect coupling. Forming DSMs constructed from execution trace data rather than source code or static analysis is also envisioned, allowing one to examine system-wide modularity properties as exhibited during runtime (helping with Q4).

Altering the normalization function could be useful in focusing the scope of the analysis on certain program elements. For example, instead of normalization evenly dividing the weights, it could give greater weight to those program elements with the scope of interest. For example, the normalization function could give greater weight to dependencies outside of a program element's package or partition, emphasizing the propagation of dependencies between separate modules in the system.

Once a DSM of the software is constructed, the above metrics can be computed quickly, allowing for the possibility of an interactive style exploration of the probabilistic metrics

both system-wide and within components and individual packages or even classes.

### 3. Related Work

Net option value analysis has been applied to DSMs [5, 8] to quantify the global value of modularity already built in to the system. It compliments our suite of metrics proposed herein, which focus on understanding the actual modularity of the system, rather than the potential benefits of modularity.

### 4. Future Directions

We have outlined new modularity metrics based on probabilistic models, but these metrics have yet to be validated or evaluated on real programs. The metrics need to be more precisely defined, and the nuances of their definitions explored, especially for programs with distinct disconnected components. Other sources for the DSM weights should be considered, such as logical coupling [3]. A full case study across multiple applications is needed in order to gain insight into how effective these metrics are for answering our motivating questions. We need to verify that utilizing a damping factor to ensure convergence does not cause major bias in the metric values. We anticipate exploring how to generate DSMs based on static analyses and runtime execution traces. In conclusion this paper is a first step towards probabilistic metrics for measuring system-wide (and subsystem-wide) modularity properties.

### References

- [1] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, 1999.
- [2] Monica Bianchini, Marco Gori, and Franco Scarselli. Inside PageRank. *ACM Transactions on Internet Technology*, 5(1):92–128, 2005.
- [3] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *ICSM '98*, page 190, Washington, DC, USA, 1998. IEEE Computer Society.
- [4] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *WWW '03*, pages 640–651, 2003.
- [5] Cristina Videira Lopes and Sushil Krishna Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05*, pages 15–26, 2005.
- [6] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: bringing order to the web. Technical Report 1999-66, Stanford University, 1999.
- [7] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *OOPSLA '05*, pages 167–176, 2005.
- [8] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE-9*, pages 99–108, 2001.