Algebraic Higher-Abstraction for Software Refactoring Automation

Iaakov Exman and Alexey Nechaev
Software Engineering Department
The Jerusalem College of Engineering – JCE - Azrieli
Jerusalem, Israel
iaakov@jce.ac.il, alosha82@gmail.com

Abstract— **Software systems losing modularity along their life cycle require refactoring to restore their understandability. However, refactoring is very complex and expensive when done at the lower abstraction source program level. This paper's message is: refactoring is both simpler and mathematically rigorous when done at a higher-abstraction level. This work proposes a novel algebraic approach to refactoring, done at a higher-abstraction level, which is then back-translated to a refactored source program level. The algebraic approach, combining the Modularity Matrix and Laplacian Matrix representations of software, has two advantages over conventional source program refactoring. First, software higher-abstractions refactoring by a spectral approach is amenable to automation. Second, the algebraic representation is a reliable source of refactoring rules, with the potential of reaching a rule set finally leading to full automation. The refactoring technique is concisely analyzed and illustrated by case studies.**

Keywords: Software Modularity; Spectral Refactoring; Algebraic Higher-Abstraction level; Modularity Matrix; Laplacian Matrix; Refactoring Rule Set; Refactoring Automation.

I. Introduction

Refactoring of legacy program code is necessary to recover desirable qualities of software modularity: viz. to enable software understanding and maintainability by software engineers. Refactoring has often been done at source code level, less frequently at a higher model level.

This work proposes that refactoring done at a higher abstraction level of software is much more productive and rigorous than the usual lower level approaches.

The higher abstraction level consists of the Linear Software Models, an algebraic theory of modular software composition, in which software systems are represented by matrices, such as a Modularity Matrix and/or a Laplacian Matrix. Spectral refactoring applied to these matrices is mathematically rigorous and amenable to automation

Beyond the advantages of the higher model level for refactoring in practice, the algebraic approach is a reliable source of refactoring rules that may potentially lead to a complete automation of refactoring.

DOI: 10.18293/SEKE2020-008

A. *Overall Algebraic Higher-Abstraction Approach*

The Algebraic Higher-Abstraction Refactoring is depicted in the right-hand-side of Fig. 1.

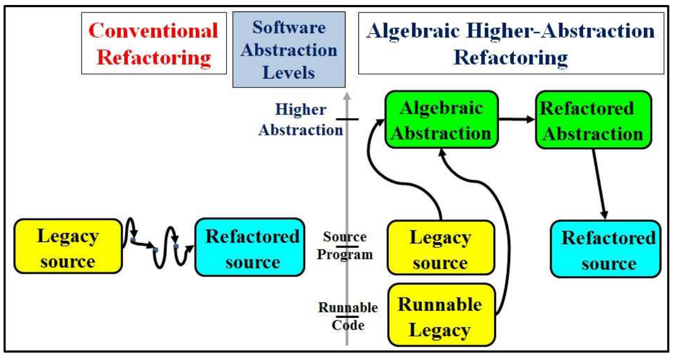


Figure 1. Algebraic Higher-Abstraction Refactoring – Conventional refactoring (left-hand-side of figure) works with complex source code. Algebraic Higher-abstraction Refactoring (right-hand-side of figure) climbs to a Higher-Abstraction level, where rigorous spectral tools resolve modules coupling in a simpler way. Its 3 stages are: 1st: source code translation into Algebraic Abstraction; 2nd: legacy abstraction modularized into a Refactored abstraction; 3rd: back-translation into Refactored source code. (All figures in color online).

Algebraic Higher-Abstraction Refactoring is simpler because higher-abstraction focuses on software architecture, ignoring source code clutter, irrelevant to modularity.

B. *Algebraic Software Modularity*

Within Linear Software Models, Modularity Matrix columns stand for *structors* – generalizing object-oriented programming classes – and rows stand for *functionals* – generalizing class methods. A 1-valued matrix element structor *provides* its functional, e.g. a class containing the declaration/definition of a method, usable by other classes. Otherwise, an element is zero-valued.

A schematic Modularity Matrix is seen in Fig. 2. It features:

- *Linear independent Matrix vectors* – true for structors among themselves and functionals among themselves;
- *Block-Diagonal Modules* – structors/functionals vectors of each module are orthogonal to other modules' vectors.

The matrix in Fig. 2 is a standard Modularity Matrix, i.e. square and without *outliers*. A Functional like F2 provided by two Structors (S2 and S3) is due to inheritance.

	S1	S2	S3	S4
F1	1	1	1	
F2	0	1	1	
F3	0	0	1	
F4				1

Figure 2. Schematic Modularity Matrix – It has 4 *structor* columns (S1 to S4), 4 *functional* rows (F1 to F4) and 2 block-diagonal modules (light blue background): a 3*3 upper-left block and a 1*1 lower-right block. Zero-valued elements outside modules are omitted for clarity.

A Laplacian Matrix is obtained from the Modularity Matrix through a bipartite graph [25]. Graph edges fit to Modularity Matrix 1-valued elements: for instance, the 1-valued (S2,F1) matrix element (Fig. 2) fits the graph edge from vertex S2 to vertex F1 (Fig. 3). Bipartite graphs have two vertex sets with edges linking only vertices in different sets.

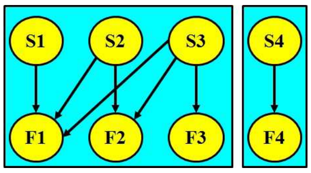


Figure 3. Bipartite Graph from Modularity Matrix in Fig. 2 – It has two vertex sets: an upper structors set (S1 to S4), and a lower functionals set (F1 to F4). Arrows pointing down mean that structors provide functionals. Rectangles (light blue) contain vertices belonging to a given module, a connected component.

The Laplacian Matrix [26] (in Fig. 4) is generated from the bipartite graph (in Fig. 3), according to equation (1):

$$L = D - A \qquad (1)$$

where ***L*** is the Laplacian matrix, ***D*** is the Degree matrix of the graph vertices and ***A*** is the Adjacency matrix of vertex pairs.

	F1	F2	F3	F4	S1	S2	S3	S4
F1	3				-1	-1	-1	
F2		2			0	-1	-1	
F3			1		0	0	-1	
F4				1				-1
S1	-1	0	0		1			
S2	-1	-1	0			2		
S3	-1	-1	-1				3	
S4				-1				1

Figure 4. Schematic Laplacian Matrix – This Laplacian is generated from the bipartite graph in Fig. 3 by equation (1). Its diagonal is the Degree matrix ***D*** (in green) displaying vertex' degrees of the Bipartite graph. The upper-right quadrant (identical to the Modularity Matrix with a minus sign) together with the lower-left quadrant is the negative of the graph Adjacency matrix ***A***.

C. *A Running Example*

We introduce a running example, to clarify the notion of outlier and its relationship to refactoring.

Outliers are 1-valued matrix elements outside of any block-diagonal modules. Outliers cause coupling of pairs of modules, which require refactoring to decouple these modules.

Fig. 5 shows a Modularity Matrix of an actual software sub-system of Intellij IDEA [15]. This matrix has two block-diagonal modules, and one outlier element coupling these two modules. Coupling means that the outlier Structor S5 belongs to the lower-right module, while its Functional F1 belongs to the upper-left module. This is an intermediate matrix: one knows the outlier location, but it must still be refactored (see section V).

Structor → Functional ↓		Javac2	Javac2 #1	uiDesigner. Ant. Javac2	Nested Form Loader	Ant.Nested Form Loader
		S1	S2	S4	S3	S5
Compile( )	F1	1	1	1		1
Accept( )	F2	0	1	0		
uiDesigner.ant. Javac2<init>( )	F4	0	0	1		
getClassToBind Name( )	F3				1	1
AntNestedForm Loader<init>( )	F5				0	1

Figure 5. Intellij IDEA Modularity Matrix with single outlier – It has 5 *structor* columns (S1 to S5), 5 *functional* rows (F1 to F5) and 2 block-diagonal modules (light blue background): a 3*3 upper-left block and a 2*2 lower-right block. There is one outlier element (F1,S5) (hatched red background) coupling the two modules. Zero-valued elements outside modules are omitted for clarity.

D. *Paper Organization*

The remaining of the paper is organized as follows. Section II mentions related work. Section III describes the Algebraic Higher-Abstraction Refactoring. The refactoring software architecture of the Re-Factory System is detailed in Section IV. Section V illustrates and analyzes refactoring by means of a few case studies. Section VI concludes the paper with a discussion.

II. RELATED WORK

This is a very concise review of the Modularity literature, due to strict space limitations.

A. *Linear Software Models*

Linear Software Models, a rigorous linear algebra theory, were developed by Exman et al. (e.g. [7], [8]), to solve the problem of software system composition from sub-systems. Software modularization by spectral methods highlights outliers coupling modules. A procedure to improve software system design is described in [9]. The Perron-Frobenius theorem (e.g. Gantmacher [14]) is central for the Modularity Matrix theory.

Exman and Sakhnini [11] generate from the Modularity Matrix a Laplacian Matrix, which obtains the same modules as the Modularity Matrix, by similar spectral methods. The Fiedler theorem [2], [13] is central to the Laplacian theory. The Fiedler eigenvector, fitting the lowest non-zero Laplacian eigenvalue, can be used to split too sparse modules and locate outliers.

B. *Alternative Modularity Analysis*

There are various less formal matrix techniques for modularity. Baldwin and Clark describe a Design Structure Matrix (DSM) in their "Design Rules" book [3]. DSM has been applied to many fields including software engineering (see e.g. Browning [5], Cai and Sullivan [6]). For alternative clustering techniques of software modules see Shtern and Tzerpos [22].

C. *Automated Modularity Refactoring*

Surveys of the multitude of papers dealing with software refactoring are found in [19], [20]. Fewer works strictly focus on automated refactoring (e.g. [23]). The work of Bavota [4] refactors software by rearranging Java packages, combining two "machine learning" methods. The paper by Zanetti [27] uses "networks theory", with probabilistic class relocation, depending on numbers of adjacent neighbors. An article by Abdeen [1] automatically reduces packages coupling and cyclic connectivity, using a "Genetic Algorithm", minimally modifying existing packages.

III. Algebraic Higher-Abstraction Refactoring

Our refactoring proposal, instead of dealing with a complex source program, or using specialized algorithms, climbs the software abstraction levels, and solves the problem in the Higher-Abstraction level with rigorous and general linear algebra. It returns to the source level the already refactored software system.

In the algebraic representation of software systems, the refactoring problem consists of recognizing each outlier which couples a pair of modules, and by relocating each outlier to a block-diagonal module, to decouple the pair of modules.

The approach essence is:

- ***Preserve overall functionality*** without change, while ***changing/creating structors***.

A. *1st Stage: Generate Laplacian and its modules*

The Algebraic Higher-Abstraction Refactoring starts from a Modularity Matrix obtained from classes/methods of a program source – the **SUD** (**S**oftware **U**nder **D**esign) – and/or its compiled code (see right-hand side of Fig. 1). The Modularity Matrix generates a Laplacian Matrix by the following steps:

- *extract a bipartite graph* from the Modularity Matrix;
- *generate the Laplacian* from bipartite graph, by eq. (1);
- *obtain module sizes and locations* from the Laplacian eigenvectors, fitting zero-valued eigenvalues;
- *split sparse modules* by the Laplacian Fiedler vector.

For the next stages (especially Back-Translation), the Algebraic Higher-Abstraction Refactoring saves the **SUD** source in a dedicated data structure: a three-columns table, whose columns are 1- functional declaration; 2- name of structor providing the functional; 3- functional implementation. The table length is the number of functionals in the program.

B. *2nd Stage: Matrices Modularization*

Next, Modularity Matrix outliers are found and decoupled.

```
Algorithm 1 – Find-&-Decouple outliers
Input:    Laplacian Matrix and its modules (from 1st Stage)
Preparation:
  Module Info Vector – saves modules size and location;
  Modularity Matrix – insert Laplacian module boundaries
                      into the Modularity Matrix;
Find Outliers: – by comparing Modularity Matrix with
                 Module Info Vector;
  Create Matrix Outlier Vector – with names of structor
         and functional containing outliers;
Decouple Outliers: – Create new Columns/Rows for
         each outlier (group) – between coupled modules;
  Single Outlier Relocation – to new column/row element;
  Outliers Group Relocation – to new columns/rows group.
Output:    Refactored Modularity Matrix & outliers.
```

C. *3rd stage: Back-Translation to Refactored Source*

Back-Translation demands challenging actions:

a) Software matrices translation – attempting to foresee every translation problem from a refactored matrix (e.g. insertion of attribute values) into source code;

b) Gradual Collection of Generic Refactoring Rules – instead of ad-hoc decisions, obtain a refactoring rule set, the basis of a future potentially complete Algebraic Higher-Abstraction automation refactoring.

The Back-Translation pseudo-code is shown next.

```
Algorithm 2 – Back-Translation to Refactored Source
                     Program
Input:   Refactored Modularity Matrix (from Algorithm 1)
Preparation: Create new source files – e.g. new .java
files;
  Insert untouched Structor Functionals into new files
    – structors from which no functionals were decoupled;
  Insert Decoupled Functionals (DF) into new files – by
    "Module Info Vector" and "Matrix Outlier Vector";
Loop: (on all DFs)        Search for DF Calls –
  Get resources used by DF – to be assembled;
  If (trivial resources) – attribute assignments as x=5,
      copy them from the original DF file, to the new file;
  Else if (non-trivial resources) – as another Function
      call, copy the "Calling-line" from the original DF file
      to the new file and adjust the relevant path, if needed;
  In any case (including no resources) – write new
  method call for the DF, in any class from which the DF
  function is called;
  Possibly use consumer matrices [12]–to find DF calls;
Output:  Refactored Source Program & its Modularity
         Matrix (for the software engineer convenience).
```

Some relevant issues are:

– ***Why create new source files instead of saving old files?***

Since the structors having decoupled functionals ought to be necessarily composed anew, it is desirable to have a single uniform way of saving the original source, i.e. by the three-columns table that was created in the 1st Stage (see sub-section *A "1st Stage: Generate Laplacian and its modules*").

D. Conjecture: Finite Refactoring Rule Set

A cardinal issue for the Algebraic Higher-Abstraction refactoring potential automation is whether the refactoring rule set is finite. Therefore we state the following conjecture.

Conjecture 1 – Algebraic Higher-Abstraction Finite Refactoring Rule Set

The number of refactoring rules in the Algebraic Higher-Abstraction Software Refactoring is finite and small.

The plausibility arguments for this conjecture are

1) The *number of refactoring types* is finite; these include:

- Single outlier – just a single unit to be relocated;
- Outliers Group – a finite small group of units to be relocated, of the order of the sub-matrix size;
- Outliers Array (sequential data) and its access functions – finite group of the order of the array size.

2) The *number of refactoring checking cases for each type referring to a group* is finite; these include:

- Direct matrix check – of the order of a sub-matrix size;
- Saved source Check – existence of different specific implementations, e.g. in an inheritance case with overridden function, of the order of the group size;
- Conceptual semantic check – where algebraic check alone is not sufficient, of the order of group size;
- Specific problems after decoupling – e.g. appearance of empty classes, of the order of the matrix dimension.

Some of these refactoring types and cases will be illustrated in section V of this paper.

IV. Re-Factory System: Software Architecture and Implementation

Re-Factory is a prototype software system designed and implemented to test case studies and the results of this work.

A. Re-Factory System: Software Architecture

The software architecture of the ***Re-Factory*** System, schematically shown in fig. 6, is composed of four sub-systems:

a- **Modulaser** – based upon an up-to-date version of this previously existing software tool [10], written in Java; inputs .class or .jar files and outputs their Modularity Matrix. In principle this tool may be adapted to deal with programs in other Object Oriented languages;

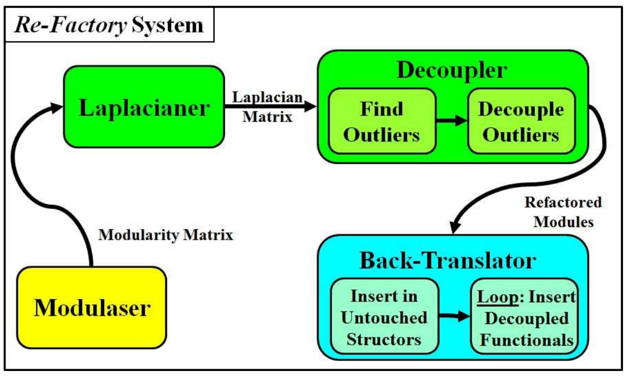


Figure 6. ***Re-Factory*** System Software Architecture – It has four sub-systems: a- an up-to-date version of the *Modulaser* tool outputs a Modularity Matrix; b- the *Laplacianer* outputs the corresponding Laplacian Matrix; c- the *Decoupler* finds and decouples outliers, then outputs **SUD** module sizes and locations; d- the *Back-Translator* outputs the refactored **SUD** modules back-translated to the source level, done in two steps: 1- inserting functionals in untouched structors; 2- Loop inserting decoupled functionals in new files.

b- **Laplacianer** – new sub-system, added to the Modulaser, generates the Laplacian and its eigenvalues/eigenvectors from the Modularity Matrix. It was extended and tested by functions of various linear algebra API libraries.

c- **Decoupler** – this sub-system has two components. One finds outliers, by direct use of Fiedler eigenvectors [13]. The other one decouples outliers using the current set of Modularity Matrix refactoring rules. More design details will be provided in an extended version of this paper.

d- **Back-Translator** – this last sub-system is also designed with two components. One of them reconstitutes the untouched structors. The other one performs a loop inserting decoupled outliers and necessary resources composing new source files.

B. Re-Factory System: Implementation

The *Re-Factory* implementation throughout the system, adopted the Modulaser Java language for compatibility. This included some frequently used API linear algebra libraries, also in Java, to calculate Laplacian eigenvalues/eigenvectors:

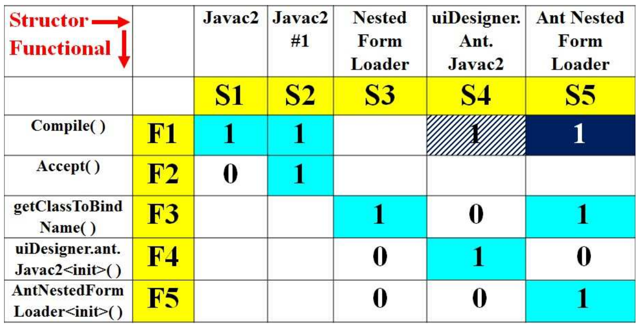
- JAMA (A Java Matrix Package) [21];
- LA4J (Linear Algebra for Java) [17];
- JBLAS (Linear algebra for Java, based upon BLAS and LAPACK) [16].

V. Case Studies: Single and Groups of Outliers

The Case Studies section illustrates and analyzes two refactoring case studies with diverse characteristics.

A. Single Outlier Refactoring

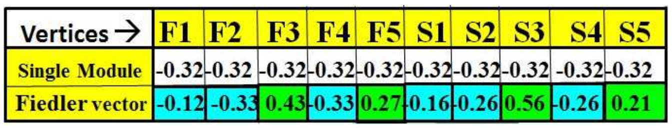
The 1st case study is a Javac2 compiler sub-system of the Intellij IDEA system [15]. This is an interesting case since the initial Modularity Matrix (Fig. 7) is puzzling: it is difficult to decide which the modules are and how many outliers are in this system. Only the Laplacian splitting resolves the puzzle.



Structor → Functional ↓		Javac2	Javac2 #1	Nested Form Loader	uiDesigner. Ant. Javac2	Ant Nested Form Loader
		S1	S2	S3	S4	S5
Compile( )	F1	1	1		▨1	1
Accept( )	F2	0	1			
getClassToBind Name( )	F3			1	0	1
uiDesigner.ant. Javac2<init>( )	F4			0	1	0
AntNestedForm Loader<init>( )	F5			0	0	1

Figure 7. Intellij IDEA Javac2 Modularity Matrix with outliers – It has two *potential* modules: one upper-left, another lower right (light blue background), whose actual sizes are not known yet. The potential modules are coupled by one or two outliers (F1,S4) and (F1,S5) (dark blue background). Coupling issues are resolved in this work by calculating the eigenvectors of the fitting Laplacian.

From the Modularity Matrix in Fig. 7 a Laplacian was generated. This Laplacian has a single zero-valued eigenvalue, thus a single whole matrix module. The Laplacian eigenvectors are shown in Fig. 8: the single module eigenvector and the Fiedler eigenvector.

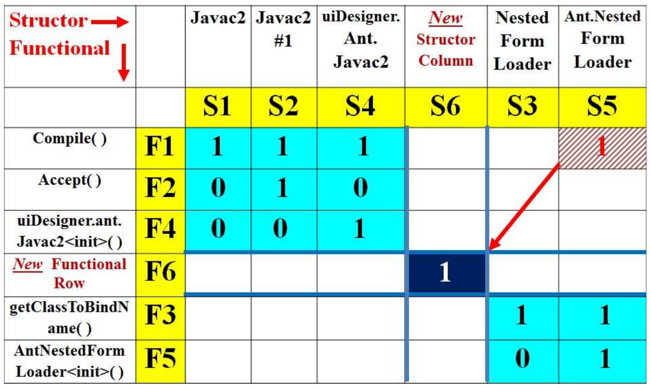


Vertices →	F1	F2	F3	F4	F5	S1	S2	S3	S4	S5
Single Module	-0.32	-0.32	-0.32	-0.32	-0.32	-0.32	-0.32	-0.32	-0.32	-0.32
Fiedler vector	-0.12	-0.33	0.43	-0.33	0.27	-0.16	-0.26	0.56	-0.26	0.21

Figure 8. Intellij IDEA Javac2 Laplacian eigenvectors – The upper row has ten vertex indices – functionals and structors – of the bipartite graph. The mid-row contains a single whole matrix module equal elements' eigenvector. The lower row shows the Fiedler eigenvector elements. It splits the single module into two smaller modules by the elements signs: negative (blue) and positive (green).

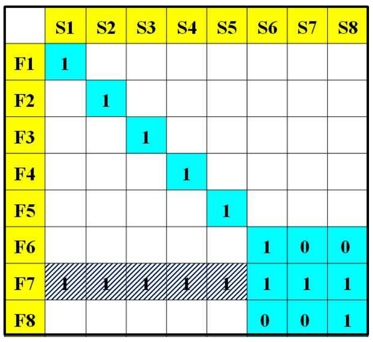
The Fiedler vector element signs split the Modularity matrix into two modules: upper-left of 3*3 size (F1, F2, F4, S1, S2, S4) and lower-right of 2*2 size (F3, F5, S3, S5). The unique outlier (F1,S5) is revealed outside both modules, as seen in the intermediate matrix (Fig. 5 in *Running Example* in section I).

Relocating the outlier (F1,S5) is now shown in Fig. 9. It has been moved to the newly created row/column (F6,S6) diagonal element. Please compare the neater refactored fig. 9 with fig. 7.



Structor → Functional ↓		Javac2	Javac2 #1	uiDesigner. Ant. Javac2	New Structor Column	Nested Form Loader	Ant.Nested Form Loader
		S1	S2	S4	S6	S3	S5
Compile( )	F1	1	1	1			▨1
Accept( )	F2	0	1	0			
uiDesigner.ant. Javac2<init>( )	F4	0	0	1			
New Functional Row	F6				1		
getClassToBindN ame( )	F3					1	1
AntNestedForm Loader<init>( )	F5					0	1

Figure 9. Refactored Intellij IDEA Javac2 Modularity Matrix – It has two modules: a 3*3 upper-left and a 2*2 lower right (light blue). The outlier (dark blue) has been relocated to a diagonal position (F6,S6) in the intersection of newly created column and row, in between the previously coupled modules. The previous (F1, S5) position is marked (hatched red).



	S1	S2	S3	S4	S5	S6	S7	S8
F1	1							
F2		1						
F3			1					
F4				1				
F5					1			
F6						1	0	0
F7	▨1	▨1	▨1	▨1	▨1	1	1	1
F8						0	0	1

Figure 10. Horizontal row case study – This matrix contains a single 8*8 big module, since the horizontal row F7 filled with 1-valued matrix elements couples all the smaller potential modules. The latter are five 1*1 diagonal modules and one 3*3 lower-right block-diagonal module (light blue background). The five horizontal matrix elements (dark blue hatched background) in row F7 are the source of the coupling problems to be solved.

B. *Outlier Group Refactoring: Horizontal Row*

The 2nd case study, the ***horizontal row*** illustrates an outlier group, with multiple usage of the same function. It is actually found in several software systems, e.g. the "Modulaser" [10] itself, and the "Tagger" software program [24]. This ***horizontal row*** example, seen in Fig. 10, may cause 3 potential problems:

a) ***Non-implemented inheritance*** – the five 1-valued row F7 matrix elements, from the left, are an outlier group to be relocated to the main diagonal. Yet, the 1-valued row of elements e.g. due to inheritance, may not be present in the original source code, except the parent class. For back-translation, an inherited but not overridden outlier function should be referred to the parent class.

b) ***Need to check source code for overridden function*** – one cannot distinguish which of the five 1-valued matrix elements were overridden by just checking the matrix.

c) ***Inability to know if coupled related tasks should not be decoupled –*** for example, the whole module in Fig.10 illustrates a Laplacianer task, computing eigenvalues and eigenvectors by differing linear algebra APIs, (see sub-section B of section IV); the five outliers perform the same task in different ways and it is not clear whether they should be decoupled. This is an example of a conceptual problem.

VI. Discussion

A. *Comparison with other Refactoring Approaches*

The case studies in section V illustrate important features of the Algebraic Higher-Abstraction Refactoring approach:

- ***Neat Representation*** – the algebraic representation of software systems by matrices clearly eliminates irrelevant source code clutter;
- ***Generic Rigorous Procedure*** – the usage of Laplacian eigenvectors for modularization is a generic rigorous mathematical procedure, avoiding ad-hoc trial and error and specialized refactoring algorithms;

- ***Exact Number of Relocations*** – no need to guess how many outlier relocations should be performed; the Fiedler vector reveals the exact number of outliers, as illustrated in the Intellij IDEA case study.
- ***Refactoring Amenable to Automation*** – the rigorous mathematical procedure, together with a finite and small refactoring rule set, is amenable to automation.

On the other hand, there still are specific problems to be solved on the way to complete automation.

B. Collecting an Algebraic Rule Set

The refactoring Rule Set collection can be seen under two perspectives: 1- ***rule classification*** into groups, as was tentatively done in sub-section D of section III; 2- ***rule conceptualization*** possibly leading to a more formal (eventually algebraic) comprehensive and self-consistent rule set. Conjecture 1 on a plausible Finite Rule Set for Higher-Abstraction supports the second perspective.

C. Algebraic and Conceptual Refactoring Separability

This research has been performed under the assumption that one can refactor software systems exclusively based upon algebraic considerations, without conceptual semantic considerations. Some case studies investigated in this work hint that the assumption is not universal. But it could still be the case that the assumption is valid in a significant majority of cases.

D. Future Work

The paper's results, in particular the conjecture of the finite refactoring rule set, deserve formal proofs and extensive verification for a variety of software systems. These will be presented in an extended version of this paper.

E. Main Contribution

This paper's main contribution is an Algebraic Higher-Abstraction refactoring, replacing conventional and less formal approaches, and amenable to software refactoring automation.

REFERENCES

[1] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil and S. Ducasse, "Towards Automatically Improving Package Structure While Respecting Original Design Decisions", in Proc. 20th WCRE Working Conf. on Reverse Engineering, pp. 212-221, (October 2013). DOI: https://doi.org/10.1109/WCRE.2013.6671296

[2] N.M.M. de Abreu, "Old and new results on algebraic connectivity of graphs", Linear Algebra and its Applications, 423, pp. 53-73, 2007. DOI: https://doi.org/10.1016/j.laa.2006.08.017.

[3] C.Y. Baldwin and K.B. Clark, *Design Rules*, Vol. I. The Power of Modularity, MIT Press, MA, USA, 2000.

[4] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk and A. De Lucia, "Improving Software Modularization via Automated Analysis of Latent Topics and Dependencies", ACM Transactions on Software Engineering and Methodology (TOSEM), Vol. 23, pp. 4, (February 2014). DOI: https://doi.org/10.1145/2559935

[5] T.Y. Browning, "Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions", IEEE Trans. Eng. Management, Vol. 48, pp. 292-306, 2001.

[6] Y. Cai and K.J. Sullivan, "Modularity Analysis of Logical Design Models", in *Proc. 21st IEEE/ACM Int. Conf. Automated Software Eng. ASE'06*, pp. 91-102, Tokyo, Japan, 2006.

[7] I. Exman, "Linear Software Models", Extended Abstract, in I. Jacobson, M. Goedicke and P. Johnson (eds.), *GTSE 2012, SEMAT Workshop on General Theory of Software Engineering*, pp. 23-24, KTH Royal Institute of Technology, Stockholm, Sweden, 2012. Video: http://www.youtube.com/watch?v=EJfzArH8-ls

[8] I. Exman, "Linear Software Models: Standard Modularity Highlights Residual Coupling", Int. Journal on Software Engineering and Knowledge Engineering, vol. 24, pp. 183-210, March 2014. DOI: 10.1142/S0218194014500089

[9] I. Exman, "Linear Software Models: Decoupled Modules from Modularity Eigenvectors", Int. Journal on Software Engineering and Knowledge Engineering, vol. 25, pp. 1395-1426, October 2015. DOI: 10.1142/S0218194015500308

[10] I. Exman and P. Katz, "Modulaser: A Tool for Conceptual Analysis of Software Systems", in Proc. SKY 2016, 7th Int. Workshop on Software Knowledge, pp. 19-26, ScitePress, Portugal, 2016.

[11] I. Exman and R. Sakhnini, "Linear Software Models: Bipartite Isomorphism between Laplacian Eigenvectors and Modularity Matrix Eigenvectors", Int. Journal of Software Engineering and Knowledge Engineering, Vol. 28, No 7, pp. 897-935, 2018. DOI: http://dx.doi.org/10.1142/S0218194018400107

[12] I. Exman and H. Wallach, "A Software System is Greater than its Modules' Sum: Providers & Consumers' Modularity Matrix", in SEKE'2019 31st Int. Conf. on Software Engineering and Knowledge Engineering, Lisbon, Portugal, pp. 75-81, July 2019. DOI: https://doi.org/10.18293/SEKE2019-003

[13] M. Fiedler, "Algebraic Connectivity of Graphs", *Czech. Math. J.*, Vol. 23, (2) 298-305, 1973.

[14] F.R. Gantmacher, The Theory of Matrices, Volume Two, Chelsea Publishing Co., New York, NY, USA, 1959. Chapter XIII, page 53, Available in the Web (out of copyright): https://archive.org/details/theoryofmatrices00gant.

[15] Intellij IDEA – IDE for Java Virtual Machine (2020). https://www.jetbrains.com/idea/

[16] JBLAS - fast linear algebra library for Java based on BLAS and LAPACK (2010) - http://jblas.org/

[17] LA4J – Linear Algebra for Java library (updated 2015) - http://la4j.org/

[18] B. S. Mitchell and S. Mancoridis, On the automatic modularization of software systems using the Bunch tool, IEEE Trans. Softw. Eng. 32 (2006) 193–208.

[19] T. Mens and T. Tourwe, "A Survey of Software Refactoring", IEEE Trans. Software Eng., Vol. 30, pp. 126-139, (2004). DOI: 10.1109/TSE.2004.1265817

[20] M. Mohan and D. Greer, "A Survey of Search-based Refactoring for Software Maintenance", J. Soft. Eng. Res. & Dev., (2018) 6:3. DOI: https://doi.org/10.1186/s40411-018-0046-4

[21] NIST, JAMA: A Java matrix package (2012), http://math.nist.gov/javanumerics/jama/

[22] M. Shtern and V. Tzerpos, "Clustering Methodologies for Software Engineering", in *Advances in Software Engineering*, vol. 2012, Article ID 792024, 2012. DOI: 10.1155/2012/792024

[23] G. Szoke, C. Nagy, R. Ferenc amd T. Gyimothy, "Designing and Developing Automated Refactoring Transformations: An Experience Report", 23rd IEEE Int. SANER Conf., Vol. 5, pp. 693-697, (2016). DOI: https://doi.org/10.1109/SANER.2016.17

[24] Tagger – software preprocessor from simple markup language to Adobe InDesign input – provided by Daniel Jackson, CSAIL, MIT – personal communication, August 2018.

[25] E. W. Weisstein, Bipartite graph (2020), http://mathworld.wolfram.com/BipartiteGraph.html

[26] E. W. Weisstein, Laplacian matrix (2020), http://mathworld.wolfram.com/LaplacianMatrix.html

[27] M.S. Zanetti, C.J. Tessone, I. Scholtes and F. Schweitzer, "Automated Software Remodularization Based on Move Refactoring", in Proc. MODULARITY '14, pp. 73-83, (April 2014). DOI: http://dx.doi.org/10.1145/2577080.2577097