Conceptual Software Design: Algebraic Axioms for Conceptual Integrity

Iaakov Exman and Phillip Katz
Software Engineering Department
The Jerusalem College of Engineering – JCE - Azrieli
Jerusalem, Israel
iaakov@jce.ac.il phillipkatz1@gmail.com

***Abstract*—Software Conceptual Integrity has been considered a cardinal concern for design and development of software systems. But, except for verbal descriptions of desiderata, there were no formal tools to guarantee that conceptual integrity is attained. This paper proposes an axiomatic algebraic approach, based upon the assumption that the software system in each level of the system hierarchy is represented by a Modularity Matrix. This representation enables to translate propriety, orthogonality and generality, usually taken as basic conceptual integrity principles, into formal algebraic criteria. The novelty is that one can finally make Conceptual Integrity quantitative calculations. The latter are illustrated by a case study. The paper also discusses the intimate relationship between software and knowledge, which emphasizes the importance of conceptual principles for software design.**

Conceptual Integrity; software system design; Linear Software Models; Modularity Matrix; axiomatic approach; formal algebraic criteria; quantitative calculations; Software Knowledge.

I. INTRODUCTION

The central importance of Conceptual Integrity for software system design and development, first advanced by Brooks [3], is a revolutionary idea, which did not percolate yet deep enough through the practice of Software Engineering. It has two fundamental implications:

a- software is far away from the machine, be it a real or a virtual machine;

b- software understanding by human or robotic stakeholders, either a developer or a user, is at the heart of high quality design of any software system.

This paper claims that faithful translation of informal verbal principles to formal mathematical criteria is essential to widespread practical application of Conceptual Integrity ideas. Our proposal formulates algebraic axioms for Conceptual Integrity based upon the Modularity Matrix [10], an algebraic structure of the Linear Software Models.

A. The Idea of Conceptual Integrity

The idea of Conceptual Integrity was proposed by Brooks [3], [4] to deal with neat design of software, and management of teams of developers of software systems.

DOI: 10.18293/SEKE2017-148

The three principles suggested by Brooks [4] and verbally described, e.g. by Jackson and co-authors [7], [17], [18] are:

1. ***Orthogonality*** – individual functions should be independent of one another;
2. ***Propriety*** – a product should have just the functions essential to its purpose and no more;
3. ***Generality*** – a single function should be usable in many ways.

These principles have not been formalized, besides being explained in natural language and illustrated by examples say by some kind of informal graphical representation.

The ideas of the Conceptual Integrity principles can be summarized in terms of design modularity and simplicity. Orthogonality is a basic mechanism to obtain modularity; propriety means an optimization i.e. the fewer the functions that perform *exactly* the same tasks, the simpler the software product; generality avoids unnecessary proliferation of *similar* functions, i.e. reuse existing functions as far as possible (but see the Discussion in section VI on faithfulness of translation).

B. Linear Software Models: the Modularity Matrix

We concisely review the Modularity Matrix, a Linear Software Models' algebraic structure, representing a software system. We aim to explain the primitive terms of these models used in the axioms of this work; for details see [8], [9], [10].

Structors – the Modularity Matrix columns – are vector characterizations of software structure, generalizing classes for any hierarchical software system level. *Functionals* – the Modularity Matrix rows – are vector characterizations of software behavior, i.e. generalized functions provided by structors that may be potentially invoked. *Modules* – the blocks along the Modularity Matrix diagonal – are sub-systems, made of structor sub-sets and their respective functionals, disjoint to sub-sets corresponding to other modules. Modules give the standard Modularity Matrix its visually recognizable block-diagonality appearance.

In Linear Software Models the design goal is to achieve for a software system, the standard Modularity Matrix, in which all blocks are orthogonal. When an outlier couples modules, which are not anymore orthogonal, the system must be redesigned.

There is a two-fold relationship between a Modularity Matrix and Conceptual Integrity: a- the structors are in fact basic *concepts* of the software system; b- the immediate algebraic translation of conceptual integrity principles such as orthogonality into operations among matrix entities.

C. Paper Organization

The remaining of the paper is organized as follows. Section II refers to related work. Section III introduces the axiomatic approach. Section IV provides quantitative Conceptual Integrity criteria enabling actual calculations. Section V illustrates the approach with a case study. Section VI concludes the paper with an overall discussion.

II. Related Work

A. Applications of Conceptual Integrity

Works that mention the importance of Conceptual Integrity often make a vague statement about its meaning, e.g. Beynon et al. [2].

Kazman and Carriere [20] deal with reconstructing a software system architecture. They use *conceptual integrity* as a guideline to a meaningful architecture, viz. use a small number of components connected in regular ways, with consistent functionality assigned to these components.

Kazman [19] describes a SAAMtool, in which *Conceptual Integrity* is estimated by the number of primitive patterns that a system uses.

Clements et al. [6] think of *conceptual integrity* as the unifying theme of the system design at all levels. Similar things should be done in similar ways, with parsimonious data and control mechanisms in a system. They refer to hierarchical levels of systems, and also suggest counting mechanisms as a path to a more quantitative definition of *Conceptual Integrity*.

A recent example of the importance of Conceptual analysis for applications of Software Engineering is the work by Zambonelli [25] on key abstractions for the Internet of Things.

B. Algebraic Representations of Software Systems

In this work we refer to the Modularity Matrix [13]. Other matrices have been used for modular design.

The Laplacian matrix – see e.g. von Luxburg [21] and references therein – has been used in various applications. Exman and Sakhnini [14] have derived a Laplacian matrix containing equivalent information about a software system as the corresponding Modularity Matrix, enabling similar software design results.

The Design Structure Matrix (DSM) has been incorporated into the 'Design Rules' by Baldwin and Clark [1]. It has been applied in various contexts, including for software systems, e.g. Cai et al. [5]. An essential difference between DSM and the Modularity Matrix is linearity as a central idea of the Modularity Matrix, while DSM design quality is estimated by an external economic theory superimposed on the DSM matrix.

Conceptual lattices, analyzed within Formal Concept Analysis (FCA) were introduced in Wille [24]. There are generic overviews describing its mathematical foundations, see e.g. Ganter and Wille [16].

The equivalence between Modularity Matrices and Conceptual Lattices – see e.g. Exman and Speicher [12] – is especially relevant for the current work, since it gives a formal justification for dealing with structors as concepts.

III. An Axiomatic Approach

In this paper we choose an Axiomatic Approach for a theory of Conceptual Software Design. This kind of approach is commonly found in certain sub-fields of mathematics and physics. Why do we need axioms here?

The first justification is that we are dealing with principles that are accepted as truth without a proof. We are unable to compact the software developer experience of Brooks, as admirable as revealed in his works, and turn it into a formal argumentation. We do acknowledge that his suggested principles are largely true, and build a model based on them.

The second justification is that we are presenting here a theory combining two independent knowledge domains, which apparently have no intrinsic relationship. Again, this relationship need to be clarified and made intuitive, but cannot be proven. One domain is the field of *Software*, having its own vocabulary and syntax – say classes, functions, inheritance, etc. The other domain is a mathematical infra-structure, which here is *Linear Algebra*, having its own vocabulary and operational rules – say matrices, scalar product, eigenvectors, etc.

This is the usual way of formulating theories in science: one chooses a mathematical formulation (here Linear Algebra) judged to be the most suitable to represent and manipulate the entities of the specific scientific domain (here Software). We need to demonstrate as far as possible that the axioms in algebraic terms are a plausible and faithful translation of the intentions of Brooks' principles referring to software conceptual integrity. This kind of demonstration goes beyond the purely algebraic arguments given under the rubric of Linear Software Models in a series of papers (see e.g. Exman [8]).

The third justification is the specific contents of the proposed axioms. These have a *prescriptive* rather than descriptive nature. The axioms embody the principles of Conceptual Integrity, which are the recommended way to design software systems as argued by Brooks [4] and other authors. Again recommended and reasonable, but not formally proven.

The ultimate justification for these axioms, which are the main contribution of this work, should be provided by rigorous empirical verification in a software design laboratory.

A. Overall Characteristics of the Axioms

The three axioms of our theory of software systems design take the verbal principles behind Conceptual Integrity as mentioned above – in sub-section *A* of the Introduction section – and represent them by linear algebraic operations within the Linear Software Models. The three axioms are presented in the logical order of the matrix manipulation needed to obtain the best system design.

B. First Axiom: Propriety

Remember the verbal formulation of Propriety: software systems should just have all the appropriate functions as demanded by its requirements, but no more than those. Its meaning is therefore to avoid superfluous functions.

> **1st Software Design Axiom: Propriety**
> For any hierarchical level of a given Software System represented by its Modularity Matrix, all its structors should be linearly independent, and all its functionals should be linearly independent.

Linear independence (of both structors and functionals) obtains exactly the desired *propriety*. Identical columns/rows in the Modularity Matrix should be eliminated, leaving just one instance of each column/row kind. If a few columns/rows are respectively linear dependent, their number should be reduced to the bare essentials, i.e. the minimal number of independent vectors.

An important algebraic consequence of the Propriety demands is that standard Modularity Matrices for any given software system are square, as shown in [10].

C. Second Axiom: Orthogonality

The verbal formulation of orthogonality is that any individual functions should be independent of one another. Its meaning is thus, no overlap at all between these functions. This formulation with "independence" appears to imply linear independence, but the latter is already achieved with the 1st axiom. Orthogonality in algebra is a stronger demand than linear independence, deserving a separate axiom.

> **2nd Software Design Axiom: Orthogonality**
> For any hierarchical level of a given Software System represented by its Modularity Matrix, all the structors/functionals belonging to a given module should be respectively orthogonal to all the structors/functionals belonging to any other module.

Orthogonality as defined in Linear Algebra, i.e. the scalar product of two vectors is zero, is indeed a stronger demand than linear independence, viz. it demands zero overlap between vectors.

Orthogonality of software system modules is made visible by reordering columns and rows in the Modularity Matrix, displaying the disjoint sets of structors and their respective functionals, the blocks along the matrix diagonal, viz. showing block-diagonality.

It seems quite clear that Brooks and other authors concerned with conceptual integrity have used the term "orthogonality" inspired by its mathematical meaning. So, it is natural to keep using exactly the same term with its perfect meaning correspondence. The specific contribution of this work is to determine which entities – which vectors – have their orthogonality calculated.

D. Third Axiom: Generality

The verbal formulation of Generality is that a function should be usable in many ways. Our interpretation is that similar functions should not be replicated in different modules, but refined into a single generic form and then possibly invoked from other modules, if necessary. In other words, in any hierarchy level of the software system, modules should be cohesive enough – functions related to one another within a module – while enabling *composition* with other modules, even from different hierarchical levels.

> **3rd Software Design Axiom: Generality**
> For any hierarchical level of a given Software System represented by its Modularity Matrix, all its modules should display high cohesion. If a module does not display high cohesion it should be split, maximizing the number of modules in the Matrix.

High cohesion in terms of modules of the Modularity Matrix means low sparsity, i.e. low numbers of zero-valued matrix elements relatively to the module size.

If a module has low cohesion it should be split by means of a procedure using eigenvectors of the Modularity Matrix (see e.g. [11]).

E. Application of Conceptual Software Axioms

The importance and practical usage of the above axioms is to obtain software system modules actually displaying Conceptual Integrity. To this end, one just needs to apply on the software system Modularity Matrix the standard operations found in Linear Software Models. These are:

1. For *propriety* – eliminate linear dependencies respectively of matrix columns/rows;
2. For *orthogonality* – reorder matrix columns/rows to obtain block-diagonality, where blocks are modules;
3. For *generality* – eliminate outliers, thereby reducing block sizes, to avoid low module cohesion, increasing the number of modules.

IV. Quantitative Criteria Calculations

Besides the formal axioms, the practical significant contribution of this work is to formulate explicit equations to *calculate quantitative criteria* to measure conceptual integrity, one criterion fitting each of the above axioms. These equations are combined to calculate criteria for a whole software system.

All the quantities involved in the calculations of Conceptual Integrity are normalized. Normalization means that these quantities are independent of the vector or matrix sizes, i.e. one divides results by relevant entity sizes.

A. Equations for Conceptual Integrity Criteria

We assume that all elements of the Modularity Matrix, and therefore of its modules, structors and functionals are non-negative. The normalized criteria are given as follows:

1. *Propriety* – Linear Dependence within a module is evaluated by equation (1), in which **r** is the rank and **c** is the number of columns of the module sub-matrix. Since module sub-matrices are square, one could use as well the number of rows instead of the number of columns. The module propriety criterion in equation (1) has a value between zero and the maximum propriety value of 1 which is obtained when **r** equals **c**.

$$\textbf{\textit{Propriety}} = 1 - ((c - r)/c) \qquad (1)$$

2. *Orthogonality* – assume a pair of vectors ***u*** and ***v*** where each of them is normalized [23], i.e. all their elements are divided by the length of the respective vector. Their Orthogonality criterion is evaluated by equation (2), where $(u \bullet v)$ is the vectors' scalar product. Orthogonality has a value between zero and the maximal value 1 obtained for zero scalar product.

$$\textbf{\textit{Orthogonality}} = 1 - (u \bullet v) \qquad (2)$$

3. *Generality* – for each module of a software system the generality criterion is given by equation (3), the *Cohesion* of the module, where *Sparsity* is the ratio between zero-valued matrix elements and the total number of matrix elements of the module. This generality expression – the module cohesion – has a value between zero and 1 and is maximal when the Sparsity is minimal.

$$\textbf{\textit{Cohesion}} = (1 - \textbf{\textit{Sparsity}}) \qquad (3)$$

B. Systems' Conceptual Integrity Calculations

Software system calculations, based upon the above equations, should be done for the whole set of Modularity Matrix modules to obtain the combined conceptual integrity criterion for the respective software system.

For instance, the orthogonality criterion of a system with ***n*** modules, where ***k*** modules are mutually orthogonal and the remaining ***n-k*** are not orthogonal, is calculated by equation (4). The measured orthogonality ***m(i)*** for each non-orthogonal module ***i*** is obtained by repeated application of equation (2). Thus,

$$\textbf{\textit{System\_Orthogonality}} = \left(k + \sum_{i}^{n-k} m(i)\right) / n \qquad (4)$$

V. ILLUSTRATION BY A CASE STUDY

A system calculation is here illustrated by a simple case study, shown in Fig. 1. It has 5 modules, with one outlier coupling between two modules (viz. modules 2 and 3), and 3 mutually orthogonal modules (viz. modules 1, 4 and 5).

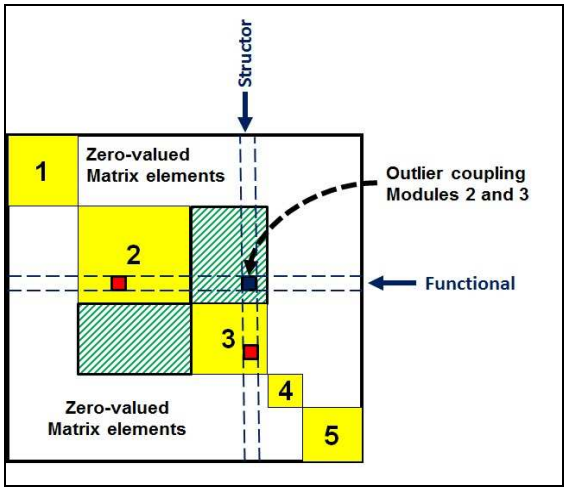

Figure 1. Schematic generic Modularity Matrix with outlier – It has 5 modules (in yellow), three of them mutually orthogonal (modules 1, 4 and 5). Modules 2 and 3 are coupled by the outlier (in blue). At least one matrix element (in red) in the coupled modules is shared by a Functional and a Structor containing the outlier. The hatched areas (in green) and the coupled modules form a larger module, too sparse to be a legitimate module, as most hatched areas matrix elements are zero-valued, except for the outlier.

The outlier couples modules 2 and 3 as it has a Functional (row) shared with module 2 and a Structor (column) shared with module 3. As seen in Fig. 1, the shared Functional has at least one non-zero matrix element in module 2. Similarly, the shared Structor has at least one non-zero matrix element in module 3. Thus, the scalar products – in equation (2) above – of the Functional and Structor containing the outlier with at least one respective Functional of module 2 and with at least one respective Structor of module 3 will be greater than zero.

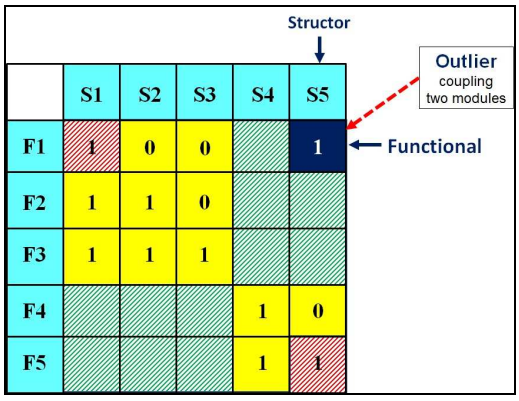

Figure 2. Specific Modularity Sub-Matrix with outlier – It is equivalent to zooming into (yellow) modules 2 and 3 of Fig. 1, whose hatched (red) elements are coupled by a (dark blue) outlier. The top-left module size is 3*3 and the bottom-right size is 2*2. The hatched areas (in green) outside the modules have all zero-valued elements (ommitted for simplicity), except the outlier.

The eigenvectors' procedure [11] obtaining Modularity Matrix modules, results in a larger module containing modules 2 and 3 and the outlier element. But, cohesion calculation of

this larger module, applying equation (3) above, shows that this larger module is too sparse to be a legitimate module. One effectively breaks it into modules 2 and 3, leaving the matrix element outside these smaller modules as an outlier.

Now let us calculate the orthogonality given by equation (4) for the software system with non-orthogonal modules shown in Fig. 2. The outlier has a functional F1 in common with the top-left module. F1 has non-zero scalar products with two other functionals (F2 and F3) of the same module. Thus, by applying twice equation (2) the average measured orthogonality ***m(i)*** for the top-left module is 0.55. In addition, the outlier has a structor S5 in common with the bottom-right module. S5 has a non-zero scalar product with the structor S4. Thus the measured orthogonality ***m(i)*** for the bottom-right module is 0.5. Therefore, the System Orthogonality calculated value is 0.81.

VI. DISCUSSION

A. The Relation between Functions and Concepts

The verbal Conceptual Integrity principles, in sub-section *A* of this paper's Introduction, are expressed in terms of functions, instead of *concepts*, as it was pointed out by Jackson and co-authors [7]. A Conceptual Integrity exposition is expected to focus on concepts. One needs to explain the role of functions. Here this is done in two steps.

In the first step, we reiterate that in the Object Oriented approach to software design, classes and their generalization *structors* stand for abstract types, i.e. actually natural language concepts. Functionals, a generalization of functions, enable the vectorial representation of Structors. In other words, they are the behavioral characterization of the respective concepts.

The second step is a reminder that the Modularity Matrix with its Structors and Functionals have been shown by Exman and Speicher [12] to be equivalent to a *conceptual lattice*. Lattices, within Formal Concept Analysis [16], are algebraic structures linking concepts of a particular domain. Thus, from this point of view, functions are relevant to conceptual analysis.

B. The Number of Conceptual Integrity Axioms

The number of Conceptual Integrity axioms is an open issue. The very formalization of the axioms should facilitate answering this question, as axioms within the formalism should display coherence. Almost paradoxically, we should apply Conceptual Integrity to decide about the number of its axioms.

It should be kept in mind, that the exact number of axioms and their specific contents are dependent on software design and development technologies. A substantial amount of automatic development and less human influence, say by deep learning techniques, would probably keep the algebraic basis for the axioms, but change specific contents.

De Rosso and Jackson [7] proposed a 4th verbal principle called *Consistency* enunciating that actions behave in a similar way irrespective of the states in which they are invoked. They did not include it among the Conceptual Integrity principles, as it seems a user interface issue rather than a deeper conceptual issue. Such verbal principle could generate a 4th axiom.

Additional axioms are certainly conceivable. But, are we sure that the minimal set is indeed three? This issue is dealt with in the next sub-section.

C. Faithfullness of Translation

The use of a mathematical formalism is never purely arbitrary. We have chosen Linear Algebra to represent software design since our analysis refers to linearly dependent information, such as conceptual dependencies, function replications and common functionalities. On the other hand, it is well known that certain science domains may be equally well represented by quite different mathematical tools. For instance, quantum mechanics was expressed through a differential equation by Schroedinger, and through matrices by Heisenberg, and it was necessary to demonstrate their non-obvious equivalence.

The issue of faithful translation of the verbal principles into formal axioms appears to be relevant to the exact number of axioms. In terms of linear algebra coherence, Orthogonality and Propriety seem to be completely justified, as they explicitly refer to linear algebra operations.

Generality, on the other hand, may raise some controversy and certainly deserves a deeper discussion. First, cohesion is a software rather than an algebraic concept, even though sparsity in equation (3) is expressed in terms of matrix notions.

Generality does not appear to be an intrinsic property of a given design of a software system. Through the representation of high-level abstraction ideas behind the software, in a broad and open sense, it rather seems to prepare the ground for *future extensions* and modifications.

D. Abstraction as a Deeper Motivation for Conceptual Integrity

The deeper motivation for the importance of Conceptual Integrity in software design and development can be inferred from the software history along time in the last fifty years. One perceives the continuous increase of the highest abstraction level from which one starts software design, with concomitant distancing away from machines, be they real or virtual.

We cautiously distinguish two kinds of abstraction hierarchies. One kind is the software system proper hierarchical levels, each level with its own Modularity Matrix. This hierarchy refers to the whole system, sub-systems, sub-sub-systems, down to indivisible components chosen by software engineers. Besides the modularity partition of the software system, it also characterizes abstraction levels. Modules in each level have a specific conceptual identity, which is more and more general as one goes up the hierarchy.

The other kind of abstraction levels is that of the implementation languages hierarchy, from the lowest machine level up to the human natural language. This hierarchy in bottom-up direction approximately passes from assembly, to high-level programming languages, to models such as UML, and finally to natural language concepts.

Higher abstraction levels imply easier grasping of the software system design by humans. Thus Conceptual Integrity is not a constraint artificially imposed from the outside, but an inherent and comprehensible quality serving human stakeholders, both the software developers and its end-users.

E. Software and Knowledge

Since this paper is presented to a conference on Software and Knowledge Engineering, it is worthwhile to stress the intimate relationship between Software and Knowledge. On the one hand, Conceptual Integrity is of cardinal importance to software system design and development.

On the other hand, concepts are first class entities in structures used to define and impart meaning to the vocabulary of a certain domain. Typical such structures are the conceptual lattices, referred above in sub-section *A* of this Discussion and their near cousins, the domain and application ontologies, which are the subject of Knowledge engineering.

F. Pragmatic Considerations

In order to apply in practice the theory exposed in this paper one needs a software tool such as Modulaser [15], enabling generation and analysis of Modularity Matrices for software systems from compiled code. One can also easily generate a Matrix from UML class diagrams, in which classes stand for structors and independent methods stand for functionals. It should be stressed that such analysis points out to coupling problems deserving system redesign, while leaving the actual solution to considerations of the software engineer.

Finally, one observes that Modularity Matrices solve a software system composition problem, somewhat similar to Petri Nets. However, the latter focuses on distributed systems, and Modularity Matrices are not specialized for these systems.

G. Main Contribution

There are two main contributions of this paper. First, the formalization of the principles behind Conceptual Integrity in an axiomatic fashion, provide a deeper justification for the algebraic operations found in the Linear Software Models.

Second, the formulation of equations finally enables calculations of quantitative criteria for Conceptual Integrity.

REFERENCES

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

[2] W.M. Beynon, R.C. Boyatt and Z.E. Chan, "Intuition in Software Development Revisited", in Proc. of 20th Annual Psychology of Programming Interest Group Conference, Lancaster University, UK, 2008.

[3] F.P. Brooks, *The Mythical Man-Month – Essays in Software Engineering* – Anniversary Edition, Addison-Wesley, Boston, MA, USA, 1995.

[4] F.P. Brooks, *The Design of Design: Essays from a Computer Scientist*, Addison-Wesley, Boston, MA, USA, 2010.

[5] 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.

[6] P. Clements, R. Kazman and M. Klein, *Evaluating Software Architecture: Methods and Case Studies*. Addison-Wesley, Boston, MA, USA, 2001.

[7] S.P. De Rosso and D. Jackson, "What's Wrong with Git? A Conceptual Design Analysis", in Proc. of Onward! Conference, pp. 37-51, ACM, 2013. DOI: http://dx.doi.org/10.1145/2509578.2509584.

[8] I. Exman, "Linear Software Models", Proc. GTSE 1st SEMAT Workshop on a General Theory of Software Engineering, KTH Royal Institute of Technology, Stockholm, Sweden, 2012a. http://semat.org/wp-content/uploads/2012/10/GTSE_2012_Proceedings.pdf.

[9] I. Exman, "Linear Software Models", video presentation of paper [8] at GTSE 2012, KTH, Stockholm, Sweden, 2012b. Web site: http://www.youtube.com/watch?v=EJfzArH8-ls.

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

[11] I. Exman, "Linear Software Models: Decoupled Modules from Modularity Matrix Eigenvectors", Int. Journal of Software Engineering and Knowledge Engineering, Vol. 25, pp. 1395-1426, 2015. DOI: http://dx.doi.org/10.1142/S0218194015500308

[12] I. Exman and D. Speicher, "Linear Software Models: Equivalence of the Modularity Matrix to its Modularity Lattice", in Proc. 10th ICSOFT'2015 Int. Conference on Software Technology, pp. 109-116, ScitePress, Portugal, 2015. DOI: 10.5220/0005557701090116

[13] I. Exman, "Linear Software Models: An Algebraic Theory of Software Composition", in Proc. 28th Int. Conf. on Software Engineering and Knowledge Engineering, Keynote Abstract, KSI Research, Redwood City, CA, USA, 2016.

[14] I. Exman and R. Sakhnini, "Linear Software Models: Modularity Analysis by the Laplacian Matrix", in Proc. 11th ICSOFT'2016 Int. Conference on Software Technology, Volume 2, pp. 100-108, ScitePress, Portugal, 2016. DOI: 10.5220/0005985601000108

[15] 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.

[16] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, Springer-Verlag, Berlin, Germany, 1998.

[17] D. Jackson, "Conceptual Design of Software: A Research Agenda", CSAIL Technical Report, MIT-CSAIL-TR-2013-020, 2013. URL: http://dspace.mit.edu/bitstream/handle/1721.1/79826/MIT-CSAIL-TR-2013-020.pdf?sequence=2

[18] D. Jackson, "Towards a Theory of Conceptual Design for Software", in Proc. Onward! 2015 ACM Int. Symposium on New Ideas, New Paradigms and Reflections on Programming and Software, pp. 282-296, 2015. DOI: 10.1145/2814228.2814248.

[19] R. Kazman, "Tool Support for Architecture Analysis and Design", in ISAW'96 Proc. 2nd Int. Software Architecture Workshop, pp. 94-97, ACM, New York, NY, USA, 1996. DOI: 10.1145/243327.243618

[20] R. Kazman and S.J. Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence." Technical Report CMU/SEI-97-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1997.

[21] U. von Luxburg, "A Tutorial on Spectral Clustering", *Statistics and Computing*, 17 (4), pp. 395-416, 2007. DOI: 10.1007/s11222-007-9033-z

[22] W. Reisig, *Understanding Petri Nets*, Springer, Berlin, 2013.

[23] E.W. Weisstein, "Normalized Vector" From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/NormalizedVector.html

[24] R. Wille, "Restructuring lattice theory: an approach based on hierarchies of concepts" In: I. Rival (ed.): *Ordered Sets*, pp. 445–470, Reidel, Dordrecht-Boston, 1982.

[25] F. Zambonelli, "Key Abstractions for IoT-Oriented Software Engineering", IEEE Software, pp. 38-45, 2017. DOI: 10.1109/MS.2017.3