Quantum Software Models: Software Density Matrix is a Perfect Direct Sum of Module Matrices

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

***Abstract*— *Quantum Software Models* is a theoretical framework to systematically design and analyze any software system – be it quantum, classical or hybrid – representing it by a design Density Matrix. Recently, we have demonstrated a top-down approach, to decompose a whole software system Density Matrix into modules, using basis vector projectors of the Matrix. However, it would be even more natural to have a systematic bottom-up procedure, to compose a whole software system Density Matrix, given a set of well-designed software module matrices, taken as sub-systems. This is exactly the paper's purpose. The result obtained: the whole software system Density Matrix is a perfect *Direct Sum* of module density matrices. This result yields clear software design benefits: it is bidirectional, one can traverse the software system hierarchy top-down or bottom-up, in particular, gradually building up the whole system from verified correct modules, assured by spectral decoupling techniques. The claim is formally validated and is illustrated by software system studies.**

Keywords*— *Software Design; Software Density Matrix; Modularity; Direct Sum; Module Density Matrix; Quantum Software.

I. INTRODUCTION

Quantum Software Models is a theory of software system design, consisting of a surprising synthesis of two apparently unrelated knowledge frames, both based upon linear algebra.

The **1st** knowledge frame, *Linear Software Models* [6], [7], a linear algebraic software design theory, where vectors stand for software concepts, combined into software system matrices, such as the Laplacian [24]. Matrices enable software ***systems decomposing into modules*** by spectral methods [8].

The unexpected synthesis starts with Frederick Brooks' original idea: "*Conceptual Integrity* is the most important consideration for software system design" [4]. *Linear Software Models* express Brooks' underlying conceptual principles in algebra: ***Propriety*** – use only absolutely necessary concepts to describe software and no more – implying vectors' *linear independence*. ***Orthogonality*** – concepts should be totally independent of one another – a stronger demand than linear independence. Propriety and Orthogonality lead to modularity.

The **2nd** knowledge frame, *Quantum theory* is a generic frame, explaining "***whole systems*** in terms of their ***parts***". It is applicable to a wide variety of physical systems and their parts, e.g. crystals, molecules, atoms, and particles.

Quantum Software Models (QSM) were inspired by *Quantum Computing,* originally proposed by the physicist Feynman [11] to perform challenging computations simulating physical ***systems composed of particles***. The QSM novelty is the application to ***whole software systems*** in terms of ***modules***.

A. From Software Modules to a Whole Software System and Back

A Schematic Transition Diagram outlining a path from separate module Density Matrices, through a Direct Sum, to a whole software system Density Matrix, and back, is in Fig. 1.

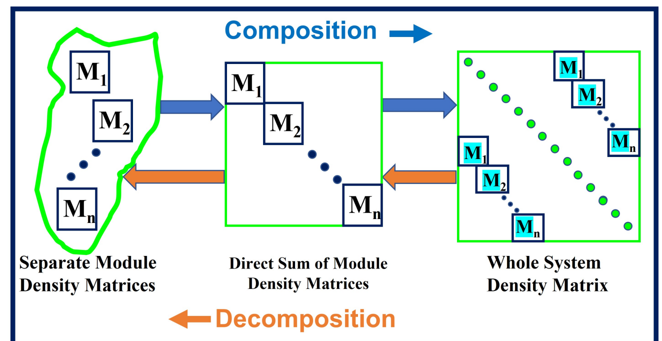

Figure 1. Schematic Transition – from separate module Density Matrices to a whole Software System Density Matrix, and back, with the intermediate modules Direct Sum. Composition is from left to right (blue arrows). Decomposition is from right to left (orange arrows). (Figures online in color).

The direct sum is formally defined for a few mathematical objects. This work only refers to matrices and subspaces *.

A matrix direct sum [20] – with symbol ⊕ – of n square matrices ($M_1, M_2, \ldots, M_n$) with possibly different sizes, constructs (see Fig. 2) a block diagonal matrix as follows

$$\oplus_j M_j = \mathrm{diag}(M_1, M_2, \ldots, M_n) \tag{1}$$

* Direct sums are also defined for mathematical "modules", which are different entities from this paper's *software modules*.

DOI: 10.18293/SEKE2022-158

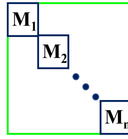

Figure 2. Matrices Direct Sum – constructs an enclosing block-diagonal Matrix.

A subspaces direct sum refers to subspaces having only the zero vector in common. As Density *Matrices* act in Hilbert *spaces* (e.g. [17] page 66), their direct sums, for software matrices and their sub-spaces, are effectively equivalent.

B. Paper Organization

The remaining of the paper is organized as follows. Section II reviews the basics of software system algebraic representation. Section III illustrates the Direct Sum composition of modules into a Classical Software System, then formulates the software modules' direct sum procedure. Section IV illustrates the procedure for a Quantum Software System. Section V validates the direct sum procedure from a few viewpoints. Section VI is a concise review of related works. Section VII concludes the paper with a discussion.

II. The Basics of Software System Algebraic Representation

A. From Bipartite Graph through Laplacian to Density Matrix

Software system algebraic design refers to 3 entities:

- *Structors* are generalizations of classes in Object Oriented Design (OOD).
- *Functionals* generalize OOD class methods.
- *Concepts* impart meaning to software systems' Structors and Functionals.

The relations between Structors labelled by (S1, S2,…,Si), and respective provided Functional declarations labelled by (F1, F2,…,Fk) are depicted in bipartite graphs [23] as in Fig. 3.

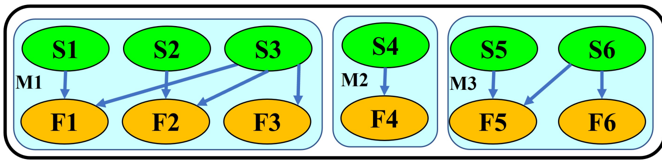

Figure 3. Bipartite Graph of a Command Design Pattern – It has 12 vertices, 6 Structors (***Si***), 6 Functionals (***Fk***), decomposable into 3 modules (***Mj***) (blue background). For instance, Structor S6 provides Functionals F5, F6 as shown by arrows. See section III-*A* for the conceptual meanings of the Command Design Pattern Structors and Functionals.

The Laplacian Matrix $\boldsymbol{L}$ is defined upon a graph as follows:

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A} \tag{2}$$

$\boldsymbol{D}$ is the diagonal Degree matrix with $\boldsymbol{D_{ii}}$ the vertex $\boldsymbol{i}$ degree, and $\boldsymbol{A}$ is the Adjacency matrix with negative 1-valued $\boldsymbol{A_{ij}}$ if vertex $\boldsymbol{j}$ is a neighbor of vertex $\boldsymbol{i}$, and zero-valued otherwise.

Von Neumann's ([22], pages 194, 214) Density Matrix ρ – the cornerstone of our *Quantum Software Models*, representing software systems – is easily obtained from a Laplacian, (Braunstein et al. [2]), by normalizing the Laplacian Matrix by its Trace (sum of diagonal degrees) $\boldsymbol{Tr(L)}$:

$$\rho = \boldsymbol{L} / \boldsymbol{Tr(L)} \tag{3}$$

B. Software Modules Represented by Density Matrices

A Density Matrix ρ may represent whole software systems or their modules. For example, the rightmost module M3 in Fig. 3, is shown again in Fig. 4, beside its Density Matrix. The Density Matrix – by eq. (3) – is the Laplacian Matrix (within parentheses) normalized by the factor 1/6.

A Density Matrix, as a normalized Laplacian, preserves all Laplacian properties, such as rows/columns summing zero, positive diagonal, and so on.

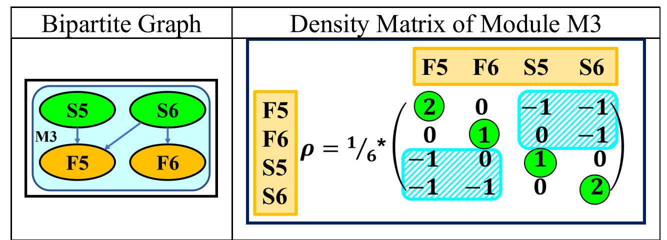

Figure 4. Bipartite Graph and Density Matrix ρ of module M3 – The left panel bipartite graph shows 4 vertices, 2 Structors labelled (**S5, S6**) and 2 Functionals labelled (**F5, F6**) as in Fig. 3. The right panel shows the Laplacian (within parentheses) {by eq. (2)}. The 4 columns and 4 rows have the same vertex order (light orange). The Degree matrix $\boldsymbol{D}$ is the diagonal (green circles). The Adjacency matrix $\boldsymbol{A}$ (hatched blue) is in the upper-right and lower-left quadrants. The Density Matrix ρ is the Laplacian normalized by the factor 1/6 {eq. (3)} since the sum of degrees of the Diagonal $\boldsymbol{D}$ equals 6.

III. From Modules to Software System: Back & Forth

This section, after an introductory classical software system, describes procedural tasks for composing a set of well-designed software modules, taken as sub-systems of a whole software system. The outcome is a formal procedure with an intermediate direct sum of modules.

A. Introductory Classical Software System: Command Pattern

Design Patterns are canonical software sub-systems to be used and re-used. Four authors, known as the *Gang-of-Four*, collected patterns in the "*GoF*" book [13].

The Command Pattern is an example, whose basic ideas are: a- it is an abstraction applicable to any common command – copy, paste, delete, save, etc. b- it has four *generic concepts*: an ***invoker***, e.g. a menu-item or a button; a chosen ***command***, one of the previously mentioned, e.g. copy; a ***receiver*** of the command application, e.g. a file or a document; a history mechanism, enabling ***undo***/***redo*** of commands perhaps invoked by mistake. These concepts, Structors & Functionals of this software system, assign conceptual meanings to the ***Si*** and ***Fk*** labels of Fig. 3. Command Pattern Structors & Functionals are shown together with their modules, in Fig. 5.

	Modules	Module Size		Structors		Functionals
M1	Command	3-by-3	S1	*I*Command	F1	Execute
			S2	*I*History	F2	Undo/Redo
			S3	Concrete Command	F3	Specify Command
M2	Invoker	1-by-1	S4	Action Invoker	F4	Invoke
M3	Receiver	2-by-2	S5	*I*File Receiver	F5	Receiver Action
			S6	Concrete Receiver	F6	Specify Receiver

Figure 5. Command Design Pattern: Structors & Functionals – It has 3 modules: Command, Invoker, Receiver. Compare with bipartite graph in Fig. 3. Structors' names with an *"I"* are Interfaces inherited by Concrete classes.

Next are transition stages from modules (Fig. 6), through a Direct Sum (Fig. 7) to System Density Matrix (Fig. 8).

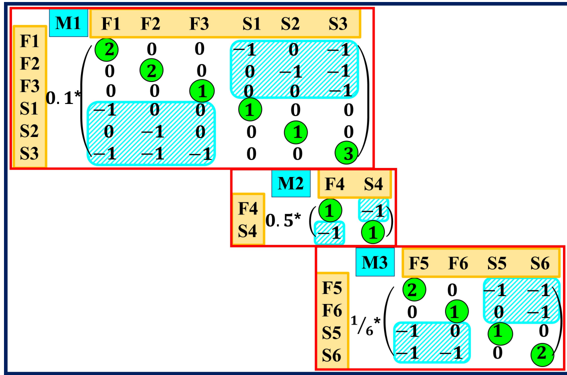

Figure 6. Command Design Pattern: Separate Modules – One sees 3 separate module Density Matrices (within red rectangles), and different normalizations: M1 has a factor 1/10, M2 has a factor ½ and M3 has a factor 1/6.

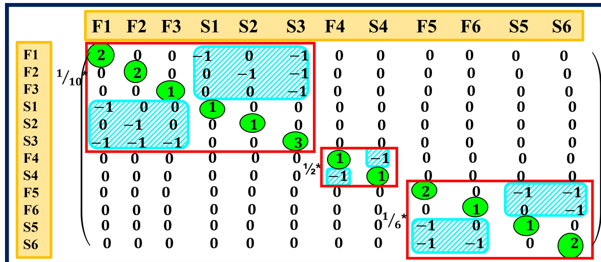

Figure 7. Command Design Pattern: Direct Sum of Module Density Matrices – The single big matrix has 3 block-diagonal modules within red rectangles.

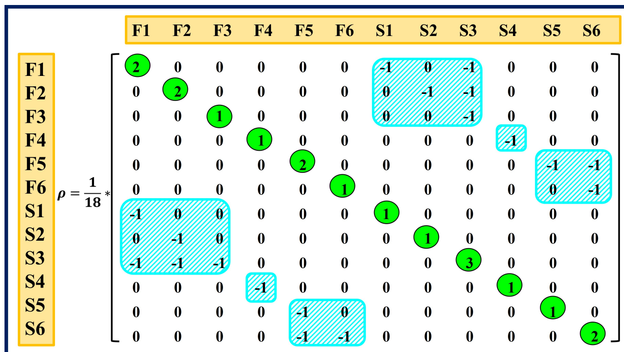

Figure 8. Command Design Pattern: Renormalized whole Software System Density Matrix – The diagonal Degree Matrix ***D*** (green circles) keeps the Direct Sum values, reordered due to repositioned Functionals and Structors. The Adjacency Matrix ***A*** (hatched blue) keeps values as each module carries the same Direct Sum columns and rows. Modules order is preserved.

Underlying the Transition from modules to the whole software system 3 tasks were performed:

1- ***Reordering Structors & Functionals*** – composition collects Structors together and Functionals together; decomposition separates them by modules;

2- ***Preserving modules order*** – the stage from Direct Sum to whole System Density Matrix, and vice-versa, keeps the same modules order;

3- ***Renormalization*** – transition through direct sum needs renormalization of density matrices in both directions.

B. Composition Procedure by Direct Sum of Modules

Here the Composition Procedure from modules to a whole software system is formulated in pseudo-code.

Composition Procedure 1– by Direct Sum of Modules
Given: Density Matrices of all Modules needed;
Obtain: Density Matrix of the whole Software System.

Preparation Phase

1. *Modules Choice* – Choose desired modules;
2. *Normalization factors* – Delete from matrices: they are not needed for composition;

Direct Sum Phase

1. *Modules' order Choice* – Decide the modules order, to be preserved in the Transition Phase;
2. *Prepare Direct Sum* – in block-diagonal format, in the decided modules' order;

Transition to Whole System Density Matrix

1. *Prepare labels list* – first Functionals, then Structors;
2. *Prepare empty system Density Matrix* – with size equal the sum of modules' Density Matrices' sizes;
3. *Loop* on list of modules, preserving their order;
   For each module Density Matrix do:
   - Add upper-right Adjacency matrix square module (including its zeros) to the system Density matrix according to the module respective row and column labels, keeping the block diagonal format of the upper-right quadrant;
   - Add diagonal degrees in the same labelled rows, such that the row values sum to zero;
   - Reflect the Adjacency matrix module, around the diagonal, to the lower-left quadrant;
   - Fill-in remaining empty system matrix elements of the whole system density matrix with zeros.
4. *Renormalize the system Density Matrix*
5. *Output system Density Matrix*

The reverse Decomposition Procedure from the whole system Density Matrix back to modules' Density Matrices is easily inferred from the above Composition Procedure.

IV. Quantum Software System And Other Systems

Having illustrated the *classical* Design Pattern, here we describe a *quantum* software system, viz. Grover Search. In addition, we concisely mention other systems.

A. Quantum Software System: Grover Search

Quantum Software systems design starts getting Structors & Functionals from Quantum Circuits ([17] page 22), i.e. sequential circuits with time increasing from left to right.

The Grover quantum Algorithm [14] searches unsorted databases with quadratic speedup relative to classical algorithms. Grover search begins with equal probability qubits superposition by the Hadamard operator ***H*** to the tensor power of ***n***, ending with measurement. Next are quantum circuit, its Direct Sum, and system Density Matrix (in Figs. 9, 10, 11).

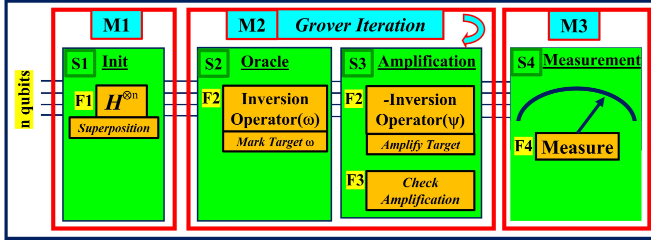

Figure 9. Grover Search Quantum Circuit – It has four (green) “boxes”, the system Structors {S1,S2,S3,S4}. Each Structor contains one or more Functionals {F1,F2,F3,F4}. The Amplification Structor S3 has two Functionals F2 and F3. Modules {M1,M2,M3} (in red rectangles) contain one or more Structors. The Grover Iteration module is a loop executed alternating the Functionals inside the Oracle S2 and Amplification S3 Structors. Check Amplification F3 decides when the loop ends, passing results to Measurement.

The *Inversion Operator* F2 of the Grover Iterator (Fig. 9) is used in both Iterator Structors. F2 has the form $\boldsymbol{I} - 2^*|x\rangle\langle x|$ ([17] page 251) with $\boldsymbol{x}$ the correct searched item value ω in the Oracle S2, or any item ψ in the Amplification S3, where F2 is multiplied by a minus sign. This is a typical inheritance case between Structors, similar to classical software inheritance.

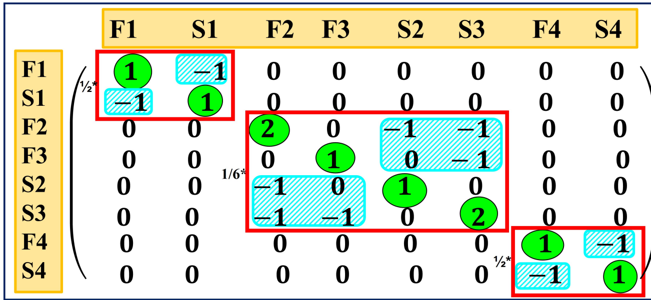

Figure 10. *Grover Search* Direct Sum of Modules’ Density Matrices – modules are block-diagonal, enclosed by red rectangles.

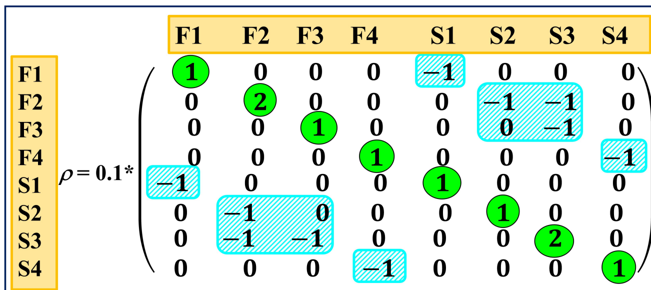

Figure 11. *Grover Search* whole software system Density Matrix.

Let us compare the Command Design Pattern – the classical software system – with the Grover Search – the quantum software system.

The diagrams serving as information source for the whole system and its modules, are different:

- UML class diagrams for classical systems;
- high-level quantum circuits, such as Fig. 9, for quantum systems.

However, once one has the list of Structors & Functionals, from then on, the procedure is identical, and totally independent of the conceptual meanings of the system.

This observation reinforces the plausibility of the idea that the same design approach should be applied to whatever kind of software system, classical, quantum or hybrid.

B. Hybrid and Simulation Systems

We have performed studies of other systems, which due to space limitations are not shown here. They will appear in a longer paper to be published. Other studies include: a) hybrid systems containing classical and quantum sub-systems; b) software systems whose purpose is not to compute specific numerical/logical results; the ultimate purpose of these software systems is to simulate real-world systems, to enable verification of the correctness of their control mechanisms, e.g. elevator systems.

V. Validation

This paper’s claim, the Software Density Matrix is a Perfect Direct Sum of Modules’ Density Matrices, and its implications are formally validated here, from three complementary viewpoints:

- The meaning of ***Perfect*** Direct Sum;
- ***Density Matrix*** as a complete information source of the Software System;
- Software ***Conceptual Integrity*** from Modularity.

A. Meaning of Perfect Direct Sum

We first provide a definition of a well-designed software system, in terms of algebraic Conceptual Integrity, in the next text-box.

Definition 1 – Well-designed Software System
A Density Matrix software system, well-designed in algebraic Conceptual Integrity terms, obeys the following Adjacency Matrix conditions, within the Density Matrix:

- *Linear Independences* – all its Structors are mutually linearly independent and all its Functionals are mutually linearly independent;
- *Square Adjacency Matrix* – each quadrant of the Adjacency Matrix, within the Density Matrix is square, a linear algebraic consequence of the previous Linear Independences’ condition [6];
- *Orthogonality* – all modules in both quadrants of the Adjacency Matrix, are block diagonal.

The meaning of the Perfect Direct Sum of the modules composing a software system is formulated in the next theorem.

Theorem 1 – *Perfect* Direct Sum of Software System Modules' Density Matrices
Assuming:
(a) each module Density Matrix of a chosen set of modules is well-designed,
and:
(b) The whole software system Density Matrix, composed of the chosen set of modules, is obtained from the Direct Sum of the modules' Density Matrices by the Composition Procedure 1;
Then:
a- The composed whole software system Density Matrix is well-designed;
b- The composed whole software system Density Matrix ***perfectly*** contains all the information of the Direct Sum, *no less and no more*, in the following sense:
1. The ***degrees'*** diagonal of the whole system Density Matrix has *exactly the same matrix elements* of the Direct Sum diagonal, only reordered;
2. The modules in the upper-right quadrant of the whole system Density Matrix have *exactly the same matrix elements* of the Direct Sum modules, in exactly the same modules' order;
3. The ***renormalization factor*** of the whole system Density Matrix – the ***inverse of the sum-of-degrees*** – is the *inverse of the sum of denominators* of the normalization factors of the Direct Sum modules.

Proof:

Item a- all vectors within each module are linearly independent by the assumption (a) of well-designed modules; modules block-diagonality follows from their matrix elements being in disjoint differently labelled columns and rows; the overall modules constitute a square, since their number of columns equals their number of rows;
Item b- 1. The degrees diagonal matrix elements are generated for all rows of each module, the same rows of the Direct Sum; they are reordered, since the columns/rows are repositioned;
Item b- 2. Exactly the same matrix elements in the same modules' order is determined by assumption (b) the Composition Procedure 1;
Item b- 3. The normalization factors' denominators are the sum-of-degrees of the respective Density Matrix. By the *Item b- 1* the degrees' of the whole system Density Matrix are exactly all the Direct Sum diagonal elements. Thus the whole Density Matrix denominator is the sum of the modules sum-of-degrees. □

B. Density Matrix: Complete Source of Software System Information

Here we regard the whole software Density Matrix as an information source about the software system, and inquire about its information completeness.

Composing the software system from modules, through the modules' Direct Sum, two information aspects need analysis:

- Completeness ***about each module*** – Theorem 1 is a full positive answer: no information is lost in the process from separate modules, through Direct Sum, to a whole system.
- Completeness ***beyond individual modules*** – the software representation of Procedure 1 is a Density Matrix for all purposes. The 1st quantum computing axiom, in von Neumann's Density Operator version ([17] page 102) is a full positive answer: "the system is completely described by its density matrix acting on the system state space".

C. Software Conceptual Integrity from Modularity

Modularity is an essential contribution to Conceptual Integrity of the whole system Density Matrix, obtained by Composition Procedure 1. This follows from assumption (a) of the *Perfect* Direct Sum Theorem 1.

Definition 1, on well-designed software systems, the basis of the referred assumption (a), explicitly incorporates the linear algebraic expressions of Frederick Brooks' underlying principles of Conceptual Integrity, viz. *Propriety* and *Orthogonality* [4].

VI. Related Works

This section offers a very concise review of related works.

We start with algebraic approaches to modularity. Within *Linear Software Models*, two spectral approaches to modularity were developed: one used the Modularity Matrix [7] and another used the Laplacian Matrix [8]. In both cases, modules were extracted from matrix eigenvectors. Within *Quantum Software Models*, Exman andrefer Shmilovich modularized software system Density Matrices [10] based upon disjoint projectors of the Matrix subspaces.

Non-algebraic alternatives have been often based upon a DSM (Design Structure Matrix) [21]. Some of them [5], were justified by economic arguments (Baldwin & Clark [1]). Another alternative, to extend UML to quantum circuits [18], is certainly interesting and accumulated large experience, but still lacks a rigorous and self-consistent theory. See also Rodriguez on OpenUP [19] and its bibliography.

Next, we refer to *Quantum Computing* (QC) (e.g. [17]) issues. This paper's Introduction states that QC is a foundation of *Quantum Software Models*. A QC modularity issue addressed by these models is human understanding and reuse of quantum circuits or parts thereof, as opposed to emphasis on efficient runtime implementation. An example [12] refers to alternative Grover iteration optimizations, with this respect.

Finally, we highlight the concepts' importance to software design. Conceptual Integrity notions and underlying principles were introduced in the well-known books by Frederick Brooks, *The Mythical Man-Month* [3] and *The Design of Design* [4]. A recent contribution to the software design field, is the book by Daniel Jackson "*The Essence of Software*", in which he explains why concepts matter for great design [15].

This, and our previous papers, claim that an algebraic approach is essential for software design, as discussed next.

VII. Discussion

A. Density Matrix Choice for Software Design

The first and foremost consideration for the Density Matrix choice for software design, is the Quantum theory in which it is embedded. It offers invaluable benefits: it is rigorous, self-consistent, and triggers otherwise unthinkable questions. Furthermore, the Density Matrix affords utmost generality relative to software systems' pure or mixed states.

Practical benefits of the algebraic design are:

a) ***Agile-Design-Rules*** –Brook's Conceptual Integrity [3][4], implemented by a Laplacian or a Density Matrix, is a theory behind Agile-Design-Rules, conforming to accepted best practices' wisdom (see [9] and its bibliography);

b) ***Input modules a priori correctness*** – the correctness of Direct Sum input modules, is assured by algebraic spectral modularization and its specialized decoupling techniques [8].

c) ***Format uniformity*** & ***Software generality*** – the core asset of the algebraic software design, is an extremely simple and uniform Density Matrix format, stimulating general applicability to software systems of any kind, size, or module numbers.

B. Direct Sum, Direct Product and Tensor Product

We use Direct Sum and not Direct Product because a direct sum element is nonzero only for a *finite* number of entries, and our Density Matrices act on *finite* vector spaces. Direct product elements may have an infinite number of nonzero entries [20]; as Lang notes for abelian groups ([16], page 36), the direct sum is a direct product subset.

The Tensor Product is applicable to Density Matrices. But its relation to Direct Sum, in the software design context, is out of the scope of this paper, and will be discussed elsewhere.

C. Future work

Two topics are the subject of future work:

1st- *Outliers' Presence* – the Direct Sum has been applied to modules' Density Matrix in the Composition Procedure 1, assuming absence of outliers coupling modules. Outliers will be pre-processed by module decoupling techniques [8].

2nd- *Overlapping concepts* – a Natural Language post-processing function will be developed, to check the existence, and to streamline semantically overlapping concepts, occurring in different modules composed by Procedure 1.

D. Main Contribution of this Paper

This paper's main theoretical contribution is the Module Matrices' Direct Sum as a perfect composition of the whole software system Density Matrix. Practical benefits are design flexibility, enabling whole system gradual build-up from verified correct modules, assured by the theoretical framework.

References

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

[2] Samuel L. Braunstein, Sibasish Ghosh and Simone Severini, "The Laplacian of a graph as a density matrix: a basic combinatorial approach to separability of mixed states", https://arxiv.org/abs/quant-ph/0406165 Oct 2006.

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

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

[5] Yuanfang Cai and Kevin 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] Iaakov 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

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

[8] Iaakov Exman and Rawi 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

[9] Iaakov Exman, "Conceptual Software: The Theory Behind Agile-Design-Rules", in Proc. SEKE'2018 30th Int. Conf. on Software Engineering and Knowledge Engineering, Redwood City, CA, USA, pp. 110-115, July 2018. DOI: 10.18293/SEKE2018-182.

[10] Iaakov Exman and Alon Tsalik Shmilovich, "Quantum Software Models: The Density Matrix for Classical and Quantum Software Systems Design", (2021) https://arxiv.org/abs/2103.13755

[11] Richard P. Feynman, "Simulating Physics with Computers", Int. J. Theor. Phys., 21:467, 1982.

[12] Caroline Figgatt, Dmitri Maslov, Kevin A. Landsman, Norbert M. Linke, Shantanu Debnath and Christofer Monroe, "Complete 3-Qubit Grover Search on a programmable quantum computer", Nature Communications, 8: 1918, 2018. DOI: https://doi.org/10.1038/s41467-017-01904-7

[13] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1995.

[14] Lov K. Grover, "Quantum Computers can search arbitrarily large databases by a single query", Phys. Rev. Lett. 79(23): 4709-4712, 1997.

[15] Daniel Jackson, *The Essence of Software*, Princeton University Press, Princeton, NJ, USA, 2021.

[16] Serge Lang, *Algebra*, Revised 3rd edition, Springer-Verlag, Berlin, 2002.

[17] Michael A. Nielsen and Isaac L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, UK, 2000.

[18] Ricardo Perez-Castillo, Luis Jimenez-Navajas and Mario Piattini, "Modelling Quantum Circuits with UML", in Proc. QSE'2021 Quantum Software Engineering Workshop. https://arxiv.org/abs/2103.16169.

[19] Andres Rodriguez, "Extending OpenUP to Conform with the ISO Usability Maturity Model", in Sauer et al. (eds.) *Human-Centered Software Engineering*, HCSE 2014, LNCS, vol. 8742, Springer, Berlin, pp. 90-107, (2014). https://doi.org/10.1007/978-3-662-44811-3_6

[20] Todd Rowland and Eric W. Weisstein, "Direct Sum", https://mathworld.wolfram.com/DirectSum.html, 2022.

[21] Neeraj Sangal, Ev Jordan, Vineet Sinha and Daniel Jackson, "Dependency Models to Manage Complex Software Architecture", Proc. *OOPSLA'05*, pp. 167-176, October 2005. https://doi.org/10.1145/1094811.1094824

[22] John von Neumann, *Mathematical Foundations of Quantum Mechanics*, New Edition, Princeton University Press, Princeton, NJ, USA, 2018.

[23] Eric W. Weisstein, Bipartite graph (2022), http://mathworld.wolfram.com/Bipartite-Graph.html

[24] Eric W. Weisstein, Laplacian (2022), http://mathworld.wolfram.com/LaplacianMatrix.html