

Article

Category-Theoretic Formulation of Model-Based Systems Architecting: The Concept→Model→Graph→View→Concept Transformation Cycle

Yaniv Mordecai ^{1,*}, James Fairbanks², Edward F. Crawley ³

- ¹ Massachusetts Institute of Technology
- ² GeorgiaTech Research Institute
- ³ Massachusetts Institute of Technology
- * Correspondence: yanivm@mit.edu
- <https://orcid.org/0000-0001-5768-1044>
- <https://orcid.org/0000-0002-1778-3350>
- <https://orcid.org/0000-0002-3006-1512>

Abstract: We introduce the Concept→Model→Graph→View→Concept (CMGVC) transformation cycle. The CMGVC cycle facilitates coherent architecture analysis, reasoning, insight, and decision making based on conceptual models that are transformed into a common, robust graph data structure (GDS). The GDS is then transformed into multiple views on the model, which inform stakeholders in various ways. This GDS-based approach decouples the view from the model and constitutes a powerful enhancement of model-based systems engineering (MBSE). CMGVC applies the rigorous foundations of Category Theory, a mathematical framework of representations and transformations. The CMGVC architecture is superior to direct transformations and language-coupled common representations. We demonstrate the CMGVC cycle to transform a conceptual system architecture model built with the Object Process Modeling Language (OPM) into dual graphs and a decision support matrix (DSM) that stimulates system architecture insight.

Keywords: Model-Based Systems Engineering; Category Theory; Object-Process Methodology; Model Analytics; Concept-Model-Graph-View-Concept; Graph Data Structures; Graph Queries; Decision Support Matrix; Matrix-Based Analysis;

Acronyms and Glossary Terms			
Acronym		Full Term	
CMGVC	Concept→Model→Graph→View→Concept	MGV	Model→Graph→View
CR	Common Representation	ML	Modeling Language
DSM	Design Structure Matrix / Decision Support Matrix	OPM	Object-Process Methodology
GDS	Graph Data Structure	OPML	Object-Process Modeling Language
MBSE	Model-Based Systems Engineering	SAM	System Architecture Matrix
		UUID	Universal Unique ID

1. Introduction

Models represent concepts by capturing the ideas, observations, notions, theories, insights, and intents that people conceive of and wish to express and communicate to others [1]. Conceptual models use syntax and semantics of a modeling language to describe the concepts. A popular example for generating a model for a concept depicts Isaac Newton’s observation of gravity, and the ensuing formulation of Newton’s Theory of Gravity, in the falling of an apple from a tree. Models represent scientific concepts, as projections of theories on phenomena; they describe and abstract phenomena, while relying on theories that may be hypothetical or partially validated [2]. Likewise, models represent engineered, man-made systems in their current or future state [3].

Models also inform concepts: we can enrich our concept, perception, and subject matter understanding, by capturing, manipulating, contextualizing, composing, and analyzing concepts and aspects. Conceptual models play a central and growingly more critical role in complex systems development and operation. Models are supposed to capture and represent various relations and

interactions, whose understanding is fundamental to comprehending solutions, designs, or operational processes. Models also have immense importance in facilitating stakeholder communication and discussion. The concept-model duo is therefore a binary self-enhancing system.

Model-Based Systems Engineering (MBSE) is the formalized application of models in systems engineering [4]. MBSE facilitates the digital transformation of systems engineering: generating and communicating digitized SE deliverables up and down the value chain.[5]. Digital systems engineering (DSE) increases the dependency on MBSE, which gradually becomes a value-focused approach, and not only a model-focused practice, and a critical asset for systems engineering in digital enterprises, as several studies show [6–9].

Generating stakeholder value out of model-based platforms is a primary expected outcome of DSE. Transforming system models into analyzable and reusable artifacts, decision-supporting information, and machine-generated insight is therefore a critical contribution to enterprise-wide DSE. A DSE ecosystem architecture defined in [10] specifies modeling, infrastructure, data services, simulation, testing, analysis, and repositories (MIDSTAR1) as primary DSE services, and management tools, interoperability services, digital representations, systems, things, auditing, and reporting services (MIDSTAR2) as primary interfaces. We shall refer collectively to such results or artifacts following model processing or analysis as *views* of those models.

Model-based views abstract, highlight, or pivot specific aspects, and become increasingly more important as models grow bigger and more complicated. Observing, reasoning about, processing, analyzing, and deciding according to such views results in ongoing concept revisions [11]. The concept-model-view triangle is therefore a self-enhancing system that extends the concept-model duo. However, the potential combination of modeling languages (MLs), models, and views explodes: if every model were to be represented by every necessary view, the number of mappings would be a product of the number of models M by the number of views V , i.e., $M \cdot V$.

In the absence of dynamic and robust model-to-view transforms, concept understanding is shaped by ML-bound views. A common representation across a wide variety of models (CR) that MLs could map into, and views could map out of, can make a difference. Graph data structures (GDS) can serve as CRs [12–15]. They are simple, robust, extendable, and amenable to a huge collection of graph-theoretic algorithms. Graph and GDS are not completely interchangeable concepts. A graph is a visual representation of a subset of the GDS, and a GDS is a logical data structure that represents relations, which can be illustrated in graphs. The term graph is used for brevity when we refer to the GDS as a mediating data structure. However, a graph visualization of any portion of a GDS, whether raw or processes, is indeed a view of the model. A robust transformation infrastructure that would transform models into GDS, and generate views from GDS, will make models and views interoperable and interchangeable, and extend the collection of views that support model analysis.

A concept-model-graph-view quadratic cyclic system is potentially more effective, extensive, and efficient than the concept-model-view trio. The evolution from concept-model duos through concept-model-view trios to **concept-model-graph-view-concept** cycles is illustrated in **Figure 1**.

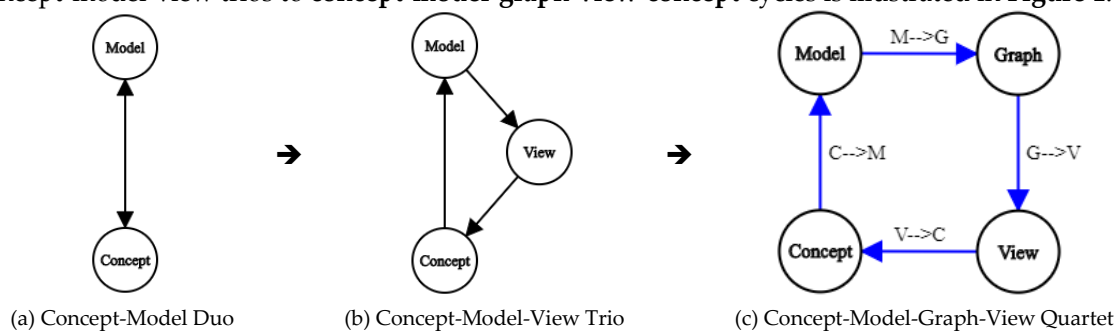


Figure 1. The Model↔Concept duo (a) is enriched with reasoning-supporting view through the Model→View→Concept trio (b) and extended as a robust Concept→Model→Graph→View→Concept (CMGVC) cycle, a computational-cognitive concept modeling and analysis process (c).

Matrices are useful views of a model. Matrices facilitate analyzing quantitative and qualitative relations. A matrix is a bidimensional discrete data structure that lends itself to various mathematical or logical analyses. Matrices are common, intuitive, and easy to work with for scientists, engineers, and analysts. As graphic MLs evolved, matrices had not found a place as formal representation modalities, although matrix-like structures or layouts remained present to some extent in modeling frameworks. For instance, the IDEF0 notation advocates a diagonal layout of blocks which resembles the N^2 matrix of inter-component dependencies; UML Activity Diagram supports a “swimlane” layout which helps divide complex multi-participant activities into columns; and Sequence Diagrams apply a similar idea, using vertically-aligned lifelines, as anchors for input and output exchange.

Simple matrices are two-dimensional and non-hierarchical. This construct is insufficient for describing complex architectures. Requirements or structural elements are leaves in an asymmetric hierarchy, which may differ in depth and detail level. Therefore, the matrix kernel must be integrated with hierarchical representations of row and column entities. Advanced matrix representations for systems engineering include the Verification Cross-Reference Matrix (VCRM), Design Structure Matrix (DSM), and System Architecture Matrix (SAM). The Department of Defense Architecture Framework (DoDAF) advocates the use of matrices in some of its views [16].

Various attempts to represent models with matrices and DSMs were mostly model-, aspect-, language-, or visualization-specific (see Section 2). A mathematically sound and robust framework for transforming models to views is needed. In fact, since models are instantiations of MLs, and modeling consists of instantiations of ML patterns, we ought to transform the ML’s set of patterns into a superset of representable data structures, rather than from a specific model to a specific view. We will focus on decision-support matrices (DSMs).

This paper defines the CMGVC transformation cycle and introduces both the cognitive and computational transforms it includes. Our framework relies on Category Theory [17]. A mathematical category consists of objects and morphisms. An Object represents a type, and a homomorphism (*hom* or morphism) is a mapping between types. The Curry-Howard-Lambek Correspondence states that categories, theories, and programming languages are equivalent, and that writing a software program is like defining a mathematical category and like proving a mathematical theory [18, 19]. This insight has inspired us to assert that MLs are also categories, and that mappings within and between modeling categories are powerful means of robust model transformations for various applications. This assertion substantiates a formal, holistic approach for perceiving and implementing concept transformations, particularly in complex systems. Such an approach is instrumental for reasoning about complex concepts using the reasoning mechanisms of people’s choice, rather than the representation mechanisms originally used for capturing those concepts. In complex systems, this idea of concept representation interchangeability is critical for a paradigm shift, cohesion of modeling and reasoning practices, and the facilitation of the rising DSE paradigm [10].

We present a series of propositions regarding the possibility, validity, and superiority of creating views from models through a robust mechanism. Our main assertion is that a transform from any model M to any view V through an intermediate GDS representation G is superior to a set of direct transformations from all models $M_1, \dots, M_M$, to all views $V_1, \dots, V_V$.

Proposition A. Superiority of GDS-Based Transform to Direct Transforms

A graph-mediated transform $M \rightarrow G \rightarrow V$ is superior to direct model-to-view transforms $M \rightarrow V$: $MGV \succcurlyeq MV$

Another important assertion is that a transform from any model M to any view V through an intermediate GDS representation G is also superior to a transform from all models $M_1, \dots, M_M$, through a ML-bound common representation CR to all views $V_1, \dots, V_V$.

Proposition B. Superiority of GDS-Based Transform to Language-Specific Transform

A graph-mediated transformation $M \rightarrow G \rightarrow V$ is superior to an ML-bound mediating representation-based transformation MRV : $MGV \succcurlyeq MRV$.

MGV is a composition of $M \rightarrow G$ and $G \rightarrow V$, so we must show that the transform $M \rightarrow G$ exists.

Proposition C. Existence of a Model-to-Graph Transform

Let ML be a modeling language and M a model in ML. There exists a formal, valid, and feasible model-to-graph transform MG from ML into GDS, $MG: ML \rightarrow GDS$.

MGV is a composition of $M \rightarrow G$ and $G \rightarrow V$, so we must show that a transform $G \rightarrow V$ exists.

Proposition D. Existence of Graph-to-View Transform

Let V be a view and G a set of GDS tuples. There exists a formal, valid, and feasible graph-to-view transform GV from GDS to V, $GV: GDS \rightarrow V$.

Specifically, with a focus on DSM views, we must show that these are valid views on GDS.

Proposition E. Existence of Graph-to-Matrix Transform

Let DSM be a decision-supporting matrix view and G a set of GDS tuples. There exists a formal, valid, and feasible graph-to-DSM transform GV from GDS to DSM, $GV: GDS \rightarrow DSM$.

With the above propositions in mind, we set out to validate and demonstrate this theory. In section 2 we review related work, concerning relevant MLs, Graph Theory, and Category Theory. In section 3 we describe the methods we use to conduct the CMGVC transformation cycle. In section 4 we assess the framework according to several criteria. In section 5 we demonstrate our method on a test case. In section 6 we discuss the results, conclude this paper and discusses potential, ongoing, and future research.

2. Related Work

In this section we review the methods, theories, and concepts that are necessary for constructing a robust CMGVC transformation cycle. We introduce Object-Process Methodology (OPM), the modeling language we focus on in this study in section 2.1. We briefly review basic concepts in Graph Theory and graph data representation in section 2.2. We discuss decision supporting matrices for systems engineering applications in section 2.3. We review Category Theory and specifically its applications and derivatives for systems engineering and MBSE in section 2.4.

2.1 Object-Process Methodology (OPM)

OPM is a leading ML and MBSE framework [4], standardized as ISO 19450 [20], which has been applied for modeling and analyzing complex and dynamic systems and processes in thousands of applications in research and industry. OPM relies on a minimal universal ontology, comprising *objects* (things that exist), which may have *states*, *processes* (things that occur), and *relations* among them. Thus, OPM's vocabulary includes about 30 terms: model, diagram, object, state, process, relation {aggregation-participation, exhibition-characterization, generalization-specialization, classification-instantiation, agent, instrument, consumption (input), result (output), effect, transformation, event (trigger), condition, invocation, in-zooming, un-folding, exception, essence {physical, informatical}, affiliation {systemic, environmental}, OPL paragraph, and OPL sentence.

OPM has a visual modality and a textual modality. The visual representation is a set of hierarchically organized Object-Process Diagrams (OPDs). All OPDs use the same notation, which means that OPM has only one kind of diagram, allowing structural, procedural, and functional aspects to appear side-by-side in any OPD. This makes system decomposition highly coherent and does not require separate notations for different levels of detail. OPM's visual modality uses rectangles to represent objects, ovals to represent processes, and few more shapes and symbols. This is irrelevant for interoperability across MLs, since the logical data representation of the model captures all the entities and relations in the model in a graphics-agnostic manner, while the graphic engine renders the visual representation according to the graphical conventions.

OPM's textual representation consists of sentences in Object-Process Language, OPL – a subset of English. Each sentence corresponds to an OPD construct – a set of linked entities – and vice versa. Each OPD is accompanied by an OPL paragraph – a set of OPL sentences. OPL is formal, machine-

readable, and color-coded. A comprehensive model-level OPL Specification (OPS) includes a set of distinct OPL sentences, such that each sentence appears in the OPS only once regardless of the number of appearances in the model (e.g. in different diagrams). Accordingly, every statement can be made only once anywhere in the model in order to hold for the entire model. Thus, modelers cannot specify two contradicting statements in two different places in the same model.

OPM has been broadly applied as a system architecting and design, stakeholder engagement and communication, and concept validation paradigm [3, 21–24]. OPM models can be easily built using OPCloud a novel cloud-based modeling studio [25]. OPCloud has been applied in medicine [26], industry [27], aerospace [28] and digital transformation [10]. OPCloud was used in this paper to build the model, generate a specification, and submit it for analysis by the proposed approach.

An OPM model of a Lane Keeping System for passenger vehicles, based on a Ford concept [29] is illustrated in **Figure 2**, including both the graphical (top) and textual (bottom) modalities.

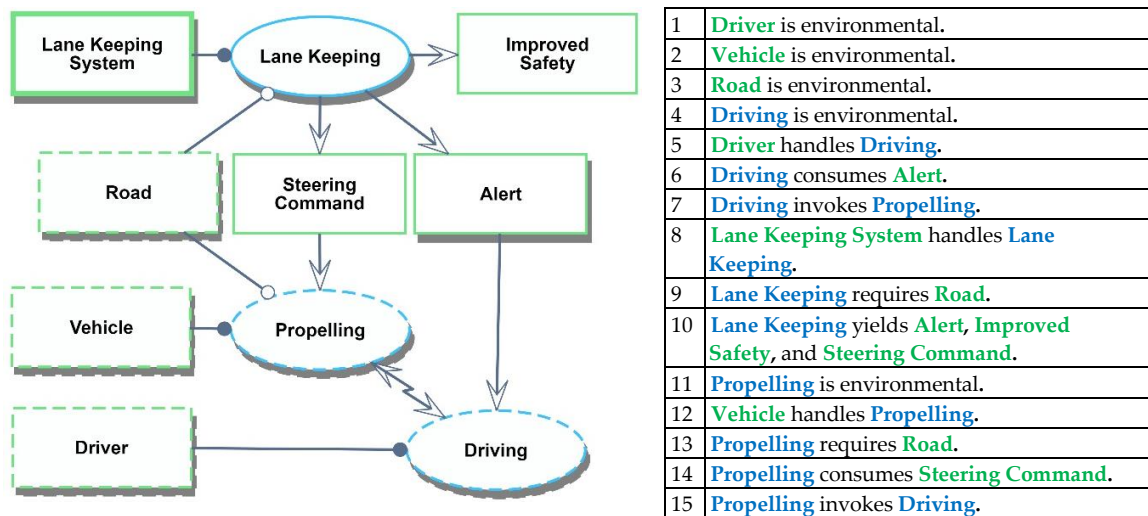


Figure 2. OPM model (OPD on the left, OPL on the right) of a Lane Keeping system, loosely based on [29]. The system handles a Lane Keeping function that provided Improved Safety. Lane Keeping generates Alerts, which inform the Driving process, and Steering Commands, which affect the Vehicle's Propelling function. The Road constitutes an instrument for both Propelling and Lane Keeping. Driving and Propelling affect each other regardless of the Lane Keeping system.

2.2 Graph Data Structures

A graph is defined as a set of nodes or vertices, with arcs or edges (commonly known as arrows) connecting pairs of nodes. In a directed graph arcs can have directions (hence the common term "arrows"), meaning that a tuple $\langle N_1, R, N_2 \rangle$ reads: " $N_1 R N_2$ ", " R connects N_1 to N_2 ", or "the direction of R is from N_1 to N_2 ". Arrows can also have types and additional attributes like values. **Figure 3** (i) shows a simple directed graph with one type of arrow and a value index.

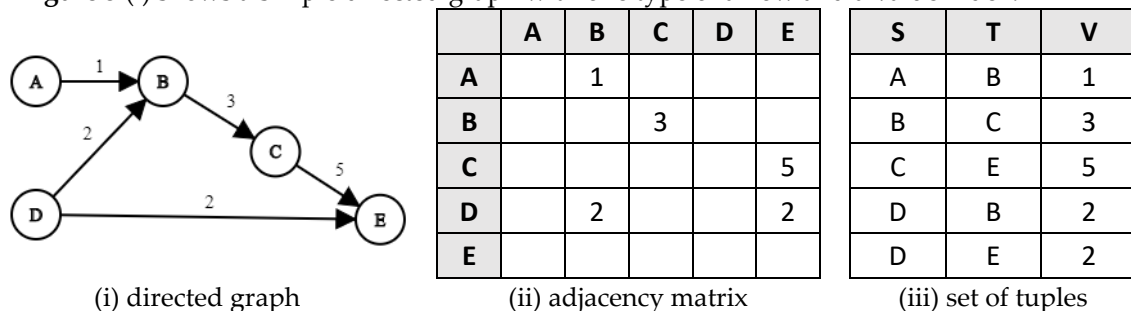


Figure 3. Equivalent representations of a set of relations between pairs of objects in a set as: (i) a directed graph, (ii) an adjacency matrix in which cell (r,c) holds the value on the arc from the object in row (r) to the object in column (c) , and (iii) a set of tuples which constitute triplets of a source object (S), a target object (T), and a value (V) that relates the source object to the target object.

Graphs lend themselves to various representations and visualizations, some of which could be matrices. **Figure 3** (ii) shows the common presentation of the graph in (i) as an adjacency matrix. The rows and columns of the adjacency matrix are the nodes of the graph, and the cells represent the values or labels of the edges. If the graph were undirected, the adjacency matrix would either have to be symmetric around the diagonal (for example, if the link from node B to node C were undirected, then the cells (B,C) and (C,B) would be both equal to 3). or triangular (cell (C,B) would be empty).

Graphs can also be represented as flat sets of tuples $\langle S, T, V \rangle$ with S as the source node, T as the target node, and V as the value, as shown in **Figure 3** (iii), or $\langle R, S, T \rangle$ tuples where R is a relation.

A set of tuples $\langle R, S, T \rangle$, potentially with additional tuple attributes like identity and value, constitutes a graph data structure (GDS). The GDS is a common exchange protocol. The Resource Description Framework (RDF) protocol for web ontology specification also consists of a set of triplets and resembles the GDS concept. RDF has also been suggested as an alternative representation of an OPM model and as the basis for textual specifications [30–32]. Indeed, GDS can be viewed as a set of RDF triplets, and vice versa. The advantage of the GDS is its ontology-indifference, and its ability to integrate the syntactic portion of the representation with the semantic one.

2.3 System Architecture and Design Decision-Supporting Matrices

The Department of Defense Architecture Framework (DoDAF) encourages the use of matrices in some viewpoints [16]. However, there are no formal requirements for the structure or layout of the matrices. In some cases, the term matrix is used to describe other tabular structures that are not matrixial in essence. For instance, a table of inputs with some attributes is not a matrix. The Unified Architecture Framework (UAF), a profile of UML, constitutes a model-based architecture framework that extends DoDAF and provides a formal modeling convention [33]. UAF advocates that aspects of the architecture specification's traceability to operational specifications, services, resources, standards, activities, etc. are represented as matrices, however there are no data structures or illustrations to demonstrate how matrices should be extracted from the profiles.

Design Structure Matrices / Dependency Structure Matrices (DSMs) are powerful tools that help systems engineers analyze various interrelated aspects, reach better comprehension, and make decisions [34]. Transforming models into DSMs is not a new idea [35–37]. DSMs require explicit definition of domain ontologies. This might be possible for syntactic ontology – the set of terms of the ML, but more difficult for semantic ontology, which reflects the modeled system or domain. Domain-specific languages (DSLs) capture domain concepts and allow for their syntactic analysis. However, DSLs may also be ambiguous. For instance, UML and SysML define multiple behavioral entities, e.g., use cases, activities, actions, message flows, and state transitions. OPM defines these as processes [4].

Multi-Domain Matrices (MDMs) extend DSMs by including multiple sets on each dimension and composing multiple matrices [38, 39]. MDMs are more informative but also more cumbersome due to their size. They also require a tremendous effort to define a) a single super-set (set of sets) of concept terms, b) a set-to-set mapping for all set pairs, and c) a member-member mapping for each pair of set members regardless of sets.

System Architecture Matrices (SAMs) are special MDMs for system architecture analysis. Technology selection, for example, can benefit from such a SAM, e.g., a quad-matrix mapping of {Systems, Processes} to {Systems, Processes} (a 2-domain MDM) [37]. The mapping relation can be defined per sub-matrix. For example: the System-System relation is an identity relation, i.e. a diagonal mapping of each system/sub-system to itself; the System-Process relation is an instrument relation; the Process-System relation is an effect relation between the Process and a System's attribute; and the Process-Process relation is an input-output transfer between the two Processes. An example of the above set of mappings is illustrated in **Table 1** (a). The mappings Instrument, Effect, Generation, and Consumption are drawn from the set of OPM relations. Additionally, the mappings Attribute, Input, Output, and ID are defined as morphisms on relations that are more straightforward to work with. For example, $\text{Attrib}(S)$ is the set of model entities E_i that participate in an exhibition-characterization relation to S: $\text{Exhibition}(S, E_i)$ or $\text{Characterization}(E_i, S)$. $\text{Input}(P_{\text{column}})$ is the set of entities E_i with an $\text{Instrument}(E_i, P_{\text{column}})$ or $\text{Consumption}(E_i, P_{\text{column}})$ relation to the column process.

A 3x3 SAM, consisting of Processes (functional and behavioral entities), Operands (inputs, resources, and outputs), and Components (structural entities), is illustrated in **Table 1** (b), based on [21]. This SAM resembles an N^2 matrix, in which the row/column items are members of the Process, Operand, and Component sets. Each sub-SAM is also a stand-alone mapping of one set to another. The cell value is the relation with which the row item relates to the column item. A row-cell-column tuple in the matrix would read: <row item (iRow)> <cell item (iRow, iCol)> <column item(iCol)>. For example: [Process P2] requires [Operand O2]; [Component C3] exhibits [Process P2].

Table 1. System Architecture Matrices

(a) 2-2 SAM mapping of systems and processes [37]

System (S)		Process (P)
System	ID(S_{row})	Instrument(S,P)
Process	Effect($P,Attrib(S)$)	Result($P_{row}, Input(P_{column})$)
		Consumption($Output(P_{row}),P_{column}$)

(b) 3-3 SAM of processes, operands, and system components [21]

	[P]	[O]	[C]
Process	[P]	[PP]	[PO]
Operand	[O]	[OP]	[OO]
Component	[C]	[CP]	[CO]

Associating set members with their corresponding set (Process, Operand, Component) is a challenge, which is not always visible at first glance. A DSL that provides a specific syntactic type for an entity simplifies identification of type-corresponding set members. However, the problem domain or solution domain may include types that exceed the DSL’s syntax. Even common terms such as Stakeholder and Operand are not included in syntactically-rich MLs like SysML [40]. Relations of hierarchy and characterization are also critical for informative SAM renditions, e.g. organization structures, activity breakdowns, product/structure breakdown, and functional decomposition.

As a broad concept, we re-purpose the acronym DSM for Decision-Support Matrix. Design structure matrices, MDMs, SAMs, pivot tables, concept maps, and a variety of matrixial and matrix-like representations, all serve a purpose of informing stakeholders, decision-makers, analysts, and other readers, to help them reason, make better decisions, and take informed action. Thus, the common goal of all those matrices is to serve as Decision Supporting Matrices – DSMs.

Generating DSMs automatically is a difficult problem. Various methods were suggested to infer DSM views from formal specifications [35, 38]. In [35] a DSM was generated out of automatically generated formal textual Object Process Language (OPL) specifications. Transforming model data into analyzable matrixial representations is not trivial. It mandates transforming any model into a uniform, common, language-agnostic representation that is robust enough to accommodate any domain-specific syntax and semantics, and powerful enough to facilitate any sort of analysis.

Pivot tables are powerful data analysis tools, typically used to aggregate, summarize, cluster, slice, and dice data. Pivot tables are embedded in common spreadsheet tools like Microsoft Excel and Google Sheets, and are also available in data analysis and business intelligence tools. The power of pivot tables lies in allowing the analyst to dynamically manipulate perspectives on the dataset. The analyst can experiment with perspectives and find informative configurations. Pivot tables rely on fact tables – flat representations of relational data. Pivot tables have been massively used in Business Intelligence applications, typically over aggregate operational datasets, often over time, geography, identity, organizational unit, etc. Pivot table applications in systems engineering are scarce [41–43], perhaps since systems engineers lack sufficient background in data analysis.

2.4 Category Theory in Systems Engineering

2.4.1 What is a category?

Category Theory is a mathematical formalism of representations and transformations that describes and deals with the structure of mathematical or mathematically-specifiable concepts. A mathematical category is an abstract mathematical structure that consists of objects and morphisms. Objects are types of entities and morphisms are functions that map objects to each other. The term object is a bit overloaded due to the use of this term in programming and engineering for instances rather than

types. OPM objects can represent both a class and an instance. We shall use the common term *Type*. A homomorphism (hom or morphism in short) is a mapping between two structures. Each morphism has a domain and codomain which are types denoted $f: X \rightarrow Y$, where X is the domain and Y is the codomain. Two morphisms can be composed if the former's codomain is the latter's domain, like functions: if $f: X \rightarrow Y$, and $g: Y \rightarrow Z$, then $f \cdot g: X \rightarrow Z$. This sequential composition executes g after f , and can also be denoted $g \circ f$. A category with three types, two explicit morphisms and one composed morphism is illustrated in **Figure 4**. A morphism can also map a cartesian product of two or more types to a new type. $f: X \otimes Y \rightarrow Z$ denotes a mapping of each combination of X and Y to Z . Z can be any structure including a multi-dimensional tuple, and the set $z \in Z$ can be a tensor.

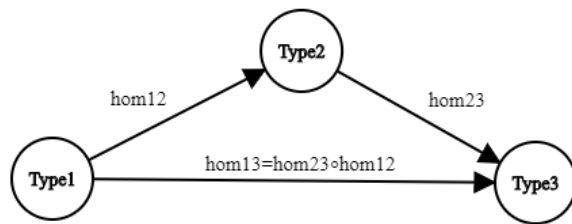


Figure 4. A category with three types: Type1, Type2, and Type3. The morphisms hom12: Type1 $\rightarrow$ Type2 and hom23: Type2 $\rightarrow$ Type3 are specified. The morphism hom13 is a composition of hom12 and hom23. We can transform Type1 to Type3 by transforming Type1 to Type2 and then Type2 to Type3.

A *functor* is a structure-preserving mapping between two categories, which maps both the types and the morphisms from one category to another. Functors can be thought of as the equivalent of functions in categorical algebra. $F: C_1 \rightarrow C_2$ denotes a functor F that transfers the types and morphisms in C_1 to types and morphisms in C_2 . Defining functors from one category to another is an important practice in establishing categorical foundations for scientific and engineering theories as it maps the concepts in a theory domain to concepts in a codomain that is possible to work with.

2.4.2 How Are Systems, Models, and Categories Similar?

The Curry-Howard-Lambek Correspondence [18, 19] asserts that programming languages and type systems are categories, in which the morphisms are rules or patterns of the language that map types to each other. A software program is an instance of a programming language that reflects or represents some real-world system. Multiple software programs may represent the same real-world system and even solve the same real-world problem in different ways. They can also be built in a variety of programming languages, and work together to solve higher-level problems and represent higher-level systems. For example, if program A assigns numbers to shapes and program B places shapes on a grid, then we can compose a program to place numbered shapes on a grid (B after A , also denoted $B \circ A$) or assign numbers to grid shapes ($A \circ B$). In this case we should expect $B \circ A = A \circ B$.

Modeling languages (ML) are programming languages, and conceptual modeling is programming [44]. System models can be thought of as instantiations of MLs, much like software programs are instantiations of programming languages. Accordingly, MLs may be thought of as categories, and in fact various system representation languages for creating graphs, matrices, trees, or animations may also be thought of as categories. Transitions across representations are essential for gaining system understanding, and for implementing the system or parts of the system. For example, transforming a functional system model into a visual animation may help explain how that system works. Mapping portions of a system into hardware and software designs is critical for implementing those parts of the system in their designated technologies. These ideas can be expressed formally using Category Theory.

Wymore defines a system as a morphism of input to output (IO), composed of a morphism of input to state (IS) and state to output (SO) [45]. A *system homomorphism* is a functional equivalence of two system designs (or models) in terms of their state-sets, their input-sets, and output-sets. A system $Z1$, with a state-set $SZ1$, an input set $IZ1$ and an output set $OZ1$, is a homomorphic image of a system $Z2$ (with respective sets $SZ2$, $IZ2$, and $OZ2$) if and only if it maps the states of $Z1$ to those of $Z2$, the inputs of $Z1$ to those of $Z2$, the outputs of $Z1$ to those of $Z2$, the input-state transforms in $Z1$ to input-state transformations of pairs to states in $Z2$, and the transformations of state to outputs in $Z2$ to transformations of states to outputs in $Z1$. Accordingly, in a category of system models, the objects are system models, and the morphisms are Wymore's system homomorphisms.

2.4.3 Category Theory applications in systems engineering, analysis, and design

Category Theory is gaining attention in recent years as a potential underlying formalism for systems engineering, as an important pillar in systems science, and as the basis for an open information modeling and analysis platform [46], with semantic roots in design theory [47]. The envisioned platform's robustness would allow for domain-agnostic cherry-picking and assembling of constructs and transforms. A roadmap for bridging the gap between theory and practice, including particular applications for systems engineering, was published by the US National Institute of Standards and Technology (NIST) [48]. State-of-the-art applications of Category Theory in systems analysis, engineering, and design include:

1. Object-Process Networks, a meta-language for System Architecting [49, 50],
2. ontological analysis based on a category *ONT* of ontologies in which Ontologies are the *types* and ontology mappings are the *morphisms* [13, 51],
3. representing a system as a category with system component types and inter-component relations as morphisms, with Ologs (knowledge representation structures that capture various non-mathematical, freely-defined relations among objects as morphisms) [52] to specify design constructs, current-state situations, constraints (the problem domain) and requirements ("shall statements") for the to-be system (the solution domain) [53],
4. representing a system as a symmetric monoidal, compact closed category with "block" types and input-output morphisms [54],
5. domain-specific reasoning in stochastic systems, mapping causal probabilistic models the category *Stoch*, whose types are measurable spaces, and whose morphisms are Markov kernels between spaces [55],
6. iterative co-design of electro-mechanical functions under a given cyber-physical system architecture [56, 57],
7. formal and verifiable system designing [58, 59],
8. Semantic modeling for extracting information from, reasoning about, augmenting, and composing computational models of complex systems and natural phenomena [60, 61],
9. hierarchical requirements engineering, gradually evolving a system architecture [62],
10. structural and functional composition of system models [63], and
11. algebraic formulation of open dynamical systems that can be characterized as resource sharers as a category of resource-sharing machines [64].

In spite of the abundance of methods, a MBSE perspective is absent. As MBSE is becoming a common and preferred way of conducting systems engineering, it is essential to harness any formalism to MBSE and to leverage MBSE through such a formalism. This paper focuses on the appropriate representation and analysis of system models that already use formal modeling languages, without replacing the existing languages with a mathematical formalism. In stead, the existing models are given artifacts and the focus is on model analysis, visualization, and delivery to various stakeholders.

2.5 Transforming Models to Graphs

Graph-centric design model representation is not new [65]. Conceptual modeling languages are inherently graphic, however graph-theoretic model analysis has not evolved alongside MLs. Recent interest can be attributed to the revival and maturation of graph technology, graph databases, and graph analytics, e.g., [66].

Mapping conceptual models to graphs remains a challenging task. Although MLs are graphical in essence, models are typically stored as logical data-structures, not straightforwardly as graphs. Graphs fail to capture the various notions that conceptual models convey, such as the distinction between artifact types and hypergraph and topological conventions (e.g. nesting, logical relations, and overlaps). Transformation rules that apply directly for the data structure used by the ML can help. For instance, UML has a uniform interchange structure, XMI, an XML structure that facilitates UML model conversions among various modeling tools. We can transform the XMI structure into a graph and analyze the model as a graph. SysML internal block diagrams can be transformed into graphs, compared against pattern graphs, and searched for the occurrence of design patterns [14].

OPM's textual machine-generated specification is another representation that is easy to read and understand for both humans and computers [20]. The textual specification covers the OPM model's logical structure. Inferring the system architecture from the model's exported report is possible, but it is not possible to reconstruct the schematic layout of the diagrams. Still, the necessary information for decision support is available. Filtering out graphical layout bias can be beneficial because the layout may affect decision-makers (for instance, due to depicting entities as central or peripheral). Conversely, layout that correctly expresses that model's intent can help in understanding the model.

Graphs belong to a class of algebraic structures known as C-sets [67]. A C-set is a functor from some category $\mathcal{C}$ into $\mathbf{Set}$, the category of sets and functions. Thus, a C-set $F(\mathcal{C})$ maps every object $c \in \mathcal{C}$ to a set $S_F(c)$, and every morphism $f_c : c \rightarrow c' \in \mathcal{C}$, to a function $f_F(f_c) : X(c) \rightarrow X(c')$. If $\mathcal{C}$ is a theory or concept, then its C-set is an instance or model of that theory. If $\mathcal{C}$ is a programming language or ML, its C-set is a projection of programs or models into $\mathbf{Set}$. One such set can be a set of tuples that represents a graph or hypergraph (with relations between more than two nodes).

Hypergraph categories are general mathematical models of hierarchy. They are applicable to various computational constructs including matrices, graphs, Markov chains, and Petri nets [67]. Hypergraph models can represent system structure, behavior, function, and interaction. Graphs of models are amenable to representation in the broader hypergraph category, an important enabler of model analysis, composition, comparison, and utilization [15].

3. Methods

This section describes a category-theoretical framework for architecting and executing the CMGVC transformation cycle. This framework consists of defining mappings between the conceptual and computational stations in the cycle: the concept phase (section 3.1), the model phase (section 3.2), the graph data structure phase (section 3.3), the view phase (section 3.4), and so on (section 3.5). While the CMGVC transformation cycle is universal, we focus on conceptual system architectures as an instance of $\mathcal{C}$, on the OPM modeling language ($\mathcal{M}$), on graph data structures ($\mathcal{G}$), and on decision-supporting matrices ($\mathcal{V}$).

3.1 The Conceptual System Architecture as a Category

Wymore's definition of a system [45], in conjunction with the conventional representation of system architecture as a combination of structure and behavior [3], gives rise to a system category, $\mathbf{SYS}$, with the following types: Structure, Input-Output (or interchangeably, Onput [68]), Resource, and State. Onputs and Resources can be referred to as Operands. The morphisms include the Behaviors, which transforms Operands, and Relations that map the types to themselves and to each other. A morphism $f : Op \otimes St \rightarrow Op \otimes St$ is a behavior of the system at any level, which transforms a combination (denoted as a Cartesian product) of operands and the system's state to another combination of operands and state. The morphism f is also a composition of the morphisms $f_{in} : Op \otimes St \rightarrow St$ and $f_{out} : St \rightarrow Op \otimes St$, i.e., $f = f_{out} \circ f_{in}$. A system sys is an instantiation of the category $\mathbf{SYS}$. Therefore, Wymore's system homomorphism is a functor $F_{Wymore} : \mathbf{SYS} \rightarrow \mathbf{SYS}$.

Cyber-physical systems (CPS) reside concurrently in the physical and cybernetic spaces, and constitute the primary concern of systems engineering. In our categorization, CPS behaviors/functions are morphisms, while system components (sensors, processors, actuators, etc.) and operands (Currency, Data, Energy, and Matter – CDEM) are types. CPS components carry out functions that convert input operands to output operands, for instance: a) an engine runs an *Energy Generating* function, to turn matter into energy, $f_{Engine} : Matter \rightarrow Energy$; b) a sensor runs a *Sensing* function that converts energy into data, $f_{Sensor} : Energy \rightarrow Data$; and c) an actuator runs an *Actuation* function that converts data into energy: $f_{Actuator} : Data \rightarrow Energy$.

Conceptual architecting usually specifies the function before the form, while physical architecting might work in the opposite direction. The attribution of a functional-behavioral morphism f_c to a structural entity $\mathcal{C}$ implies or entails a relational morphism $r_{f \rightarrow c} : f_c \rightarrow \mathcal{C}$ that assigns a function to a function-performing system component, and a dual $r_{c \rightarrow f} : \mathcal{C} \rightarrow f_c$ that

allocates a component to perform a function. This mapping of morphism to type and type to morphism implies a type-morphism duality of system behavior.

CPS functions can be composed to create higher-level functionalities, for example: sensing, then actuating: $Matter \rightarrow Energy \rightarrow Data \rightarrow Energy$. We can also define functions as Cartesian products, e.g. any plant's production function can be defined as $(f_{plant}: Data \otimes Matter \otimes Energy \rightarrow Data \otimes Matter \otimes Energy)$, which means that input data (commands), matter (raw material), and energy (e.g., electrical power) are converted to output data (statuses, reports), matter (finished goods, byproducts), and energy (e.g., heat). Some sequences may not be valid within particular system domains. For instance, in logistics, manufacturing a device (an actuator or sensor) is possible, but in an operational system such as an aircraft or autonomous vehicles, manufacturing a new part may not make sense. On the other hand, sensor "manufacturing" of images from signals may be valid.

Our categorical formulation extends the notion of a system as a procedural entity to a behavior-enabling structure, which affects operands. The operands are also types. In fact, a system can be an operand and an operand can be a system. Imagine a vehicle exiting the production line as an operand, and emerging as a system, or a software functionality delivered as a file, i.e., as an operand, and emerging as an executable, i.e. a morphism.

3.2 The Modeling Language as a Category

This paper asserts that MLs are categories, since MLs are programming languages [44], and programming languages are categories [18]. We must still specify a valid categorical structure for each ML. MLs like OPM and SysML [69] capture structural, behavioral, and relational entities syntactically. The concept set constitutes a syntactic domain-agnostic ontology that accommodates a wide range of instantiations. The *Block*, for instance, is a fundamental SysML concept. SysML Blocks are modeled in Block Definition Diagrams (BDDs) and Internal Block Diagrams (IBDs). BDDs capture relations among blocks, while IBDs capture interactions among blocks and their internal structure. A block may be both structural and functional, which can be both useful and confusing. For example, a block can specify both the sensor and its sensing function, f_{sensor} .

In SysML and its predecessor, the Unified Modeling Language (UML) [70], capturing a component's behavior, rather than its structure or function, requires behavioral notation such as Activity Diagram or State Chart. In the Activity Diagram notation activities and actions are 'boxes' (types), while control or data flows are 'arrows' (morphisms). In the State Chart notation states are 'boxes' and state-transitions are 'arrows'. The different semantics applied to boxes and arrows in SysML diagrams can be quite confusing, although experienced analysts know which notation to use to interpret various diagrams to make sense. Transforming SysML models through categorical specifications into some unified, coherent notation requires tremendous effort. Due to its scale, we defer this endeavor to future research.

Conversely, OPM models use a minimal ontology of objects, processes, and relations, in which representation ambiguity and redundancy are eliminated or minimized. Since OPM's fundamental building blocks are objects that represent structure and processes that represent behavior, with a relatively small set of relations among them, a category of the OPM language is likely to be small and handy. The challenge is in correctly representing the language domain. One apparent categorical specification could define an OPM category with objects as types and processes as morphisms.

Types in one category can be morphisms in another category. OPM makes no syntactic distinction between component-representing objects and operand-representing objects. In fact, the same object can be both an operand of one system and the operator in another. Therefore, objects can be morphisms if they perform processes, and operands if they feed into or out of processes. Processes and relations represent system morphisms but are captured as graphical shapes, which may be puzzling. Processes and relations also have attributes of their own (processes have durations, relations have cardinalities, etc.).

Any model element is a type, including those that might appear to have operational semantics, such as processes, functions, transitions, etc. This also makes sense from a visual perspective: each

shape on the canvas must be explicitly defined, regardless of its geometry or semantics. Thus, both 'boxes' and 'arrows' must be captured in MLs as types.

If operational entities are types, rather than morphisms – then what are the morphisms? Every model has a logical layer that underlies the visual layer. Therefore, every model element must be captured logically, otherwise it is no more than a meaningless sketch. It follows that there exists a hidden type, Specification (Spec) that captures the logical representation of the visual model. Thus, *specifying* – mapping visual elements to Spec – is a morphism.

However, there is another kind of morphism: *modeling* – which leads us through model creation or transformation. As modelers, we can take various modeling steps based on the state of our mode: a) start with a blank diagram; b) add an object, or process – transforming our diagram into a diagram with an object or a process – Hence the morphisms $\text{Diagram} \rightarrow \text{Object}$ and $\text{Diagram} \rightarrow \text{Process}$ exist; c) create a new diagram to specify the details of an object or process – hence the morphisms $\text{Object} \rightarrow \text{Diagram}$ and $\text{Process} \rightarrow \text{Diagram}$ exist; d) add a state to an object: $\text{Object} \rightarrow \text{State}$; or e) add a relation between an entity (object, process, or state) and another entity, which results in a two-step morphism: $\text{Entity} \rightarrow \text{Relation}$ and $\text{Relation} \rightarrow \text{Entity}$. OPM does not permit creating a Diagram from a State or Relation – although such an extension can be theoretically valid and possible. Challenging the existence or absence of a modeling morphism may help enhance the ML.

The manual modeling operations on a canvas are realized by computational commands behind the scenes, or directly through a programming interface. This allows for automating the modeling process and for creating models based on other formal representations without manual editing. Although to the best of our knowledge such programmatic commands are not available in OPM, this can be a future extension based on categorical definitions as defined here.

Reconstructing a model from its specifications gives rise to a third morphism to transform Spec statements into visual geometries or textual/logical reports: *rendering*. The pair *specifying*—*rendering* is isomorphic if a Spec can facilitate complete reconstruction (rendering) of the model. OPML also renders part of the Spec as a textual OPL paragraph – OPM's textual modality.

The notion that *modeling*, *specifying*, and *rendering* are morphisms in a ML category is illustrated in **Figure 5** for the OPM ML Category, *OPML*. The Spec is a critical type, as it enables reconstructing, composing, transforming, and comparing models. Rendering the Spec as a processable mediating artifact that represents the model substantiates a mapping of the model to another representation – a functor that maps the ML category to another category. OPCloud's Exported Report of the OPM model constitutes such a mediating artifact.

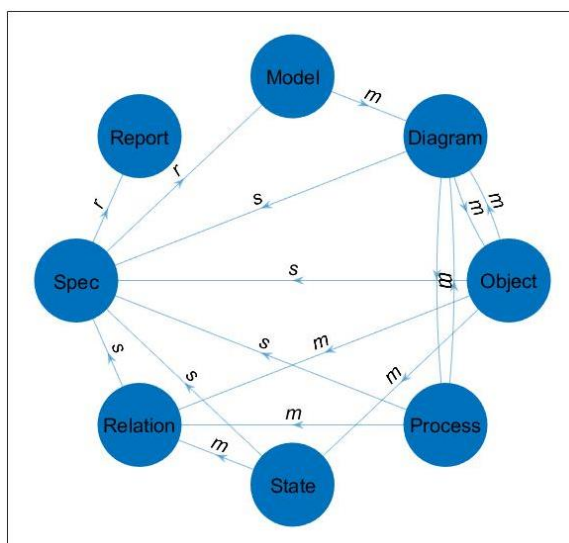


Figure 5. The OPM modeling language, OPML, as a category. Modeling (the morphism m) is like hopping through the instantiation of category types based on existing artifacts. For example: a) creating a new Diagram D1 in a Model M, b) creating an Object Ob1 in D1, c) creating a Process P1 in D1, d) creating a RelationRr1(Ob1,P1) from Ob1 to P1, e) creating a new Diagram D2 from P1, f) adding States St11, St12 to Ob1 in D2, and so on. Every instantiation of a type also affects the Spec, and therefore all graphical elements have a mapping into Spec (the morphism s). Spec also feeds into the Rendering morphism r , which renders a Report from the Spec, currently in the form of a PDF file. Diagram created using MATLAB 2019b digraph plotting capabilities [71].

Much like the Report is the model's interface to the digital ecosystem, model entities are its interfaces to the System category, *SYS*. Model, Diagram, and Spec are not real System concepts – they merely help us manage system complexity. This System-to-Model mapping constitutes the $C \rightarrow M$

portion of the CMGVC cycle. The $OPML \rightarrow GDS$ transform functor maps all the types in OPML to GDS tuples. This $C \rightarrow M \rightarrow G$ segment of the CMGVC cycle is illustrated in Table 2.

Table 2. Mapping the category of Systems, *SYS* to the category of OPM models, *OPML*, and to the category of Graph Data Structures, *GDS*

Category	Systems, <i>SYS</i>	$C \rightarrow M$	Object-Process Modeling Language, <i>OPML</i>	$M \rightarrow G$	Graph Data Structure, <i>GDS</i>
Types			Model, Diagram, Spec		
	Structure	$\rightarrow$	Object		
	Relation	$\rightarrow$	Process, Relation		
	State	$\rightarrow$	State		
	Operand (Input,Output)	$\rightarrow$	Object		
Morphisms			Report	$\rightarrow$	Tuple set
	Behavior	$\rightarrow$	Process		
	N/A		modeling, specifying, rendering		tuple operation

3.3 Transforming Models to Graphs

The MGVC transform is a composition of a transform from a model into a uniform GDS representation ($M \rightarrow G$), with a second transform from GDS into view ($G \rightarrow V$). Transforms into GDS require mapping each ML construct to type in the GDS category. Graphs have only two elements: *nodes* and *edges*. GDS has only one element: *tuple* (an abstraction and extension of a node-edge-node triplet). A graph is merely a view on GDS. Such transforms are possible on graphic MLs, although graphics do not always map to simple graphs – for instance, when shapes are contained inside other shapes or overlap with other shapes (as in Venn diagrams). OPcloud’s Exported Report includes the textual OPL specification and lists of all the objects, processes, and relations in the model. This helps in mapping all model entities and relations and translating them to GDS tuples. The final product G is a set of tuples, which includes a relation R, a source node S, a target node T, a unique identifier U, and a valuator V, denoted as the RSTUV tuple. V is useful for various applications such as quantification, verification, validation, versioning, and configuration control.

Let us revisit the lane keeping system model in Figure 2 and study its transformation into a GDS. We illustrate the same model as a graph, as shown in Figure 6.

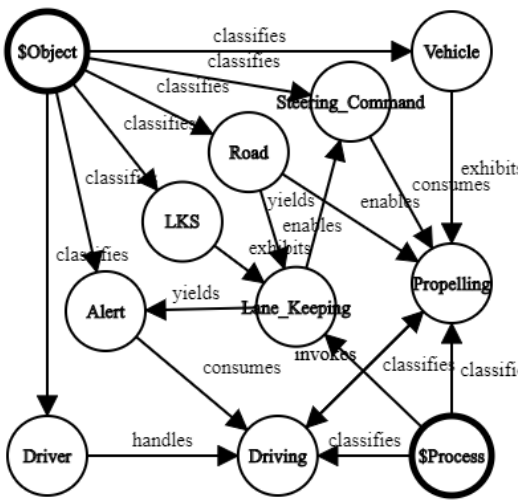


Figure 6. A graph representation of a subset of the Lane Keeping System (LKS) model in Figure 2. The graph maps the OPM-specifiable terms “\$Object”, and “\$Process” to Domain entities: Object classifies Vehicle, Driver, and LKS; Process classifies Driving, Propelling and Lane_Keeping. Domain-internal relations include the mutual invocation between Driving and Propelling, output of Alert and Steering_Command to Driving and Propelling, respectively, and the exhibition of Lane_Keeping by LKS. The graph is a visualization of a subset of the GDS that represents the model. It is clearly already difficult to comprehend, and is illustrated to build intuition about the possible mapping of a model to a graph.

RSTUV is sufficient for representing model data. It even captures aspects that are not specifically illustrated in the model. For example, it specifies the model’s inclusion of the diagrams, as well as

each diagram's inclusion of visualized objects and processes. ML terms are also defined as RSTUV identity tuples, in which $R=Identity$, $S=entity$, and $T=U=UUID$. The set of RSTUV tuples is obtained by executing the following functorial mappings from *OPML* to *GDS*:

- All entities are mapped to their OPM entity type (model, diagram, object, process, etc.) using a Classification relation. $R=Classification$, $S=E_i$, $T=OPM.Classification(E_i)$.
- All relations in the model are mapped such that $R=OPM.Relation(E_i, E_j)$, $S=E_i$, $T=E_j$, such that E_i and E_j are defined in the model as connected by the relation R .
- All entities are mapped to each diagram that includes them through an Inclusion relation. $R=OPM.Inclusion(D_i, E_j)$, $S=D_i$, $T=E_j$, where D_i is a diagram and E_j is an entity contained in D_i .
- All relations are mapped to each diagram that includes them through an Inclusion relation. $R=Inclusion$, $S=D_i$, $T=R_k$, where D_i is a diagram and R_k is a relation contained in D_i .
- All categorizations of the affiliation of an entity (as either systemic or environmental) and the essence of an entity (as either physical or informatical) [4] are mapped as Affiliation and Essence relations with the entity as the source and the designated affiliation/essence category as the target. $R=OPM.Affiliation$, $S=E_i$, $T=OPM.Affiliation(E_i)$, and $R=OPM.Essence$, $S=E_i$, $T=Essence(E_i)$.

According to Category Theory, each object must be connected to itself through an identity morphism [52]. Thus two more mappings are required:

- All entities are mapped to universally-unique identification numbers (UUIDs) through an Identity relation that constitutes the identity morphism of each entity onto itself. $R=Identity$, $S=E_i$, $T=UUID(E_i)$, where E_i is an entity in the model and $UUID(E_i)$ is its unique identifier.
- All relations are mapped to UUIDs through an Identity relation that constitutes the identity morphism of each relation onto itself. $R=Identity$, $S=R_k$, $T=UUID(R_k)$, where R_k is a relation in the model and $UUID(R_k)$ is its unique identifier.

The $OPML \rightarrow RSTUV$ functor that executes the above mappings is illustrated in **Figure 7**. The functor extracts the OPM elements from the report generated by OPCloud (*OPM.ExportedReport*). *ExportedReport* provides most of the information needed for analyzing an OPM model. It does not include shape positions in the diagrams, and therefore, it is not possible to reconstruct the diagrams with the shapes' exact positioning, but only up to the participation of elements in each diagram.

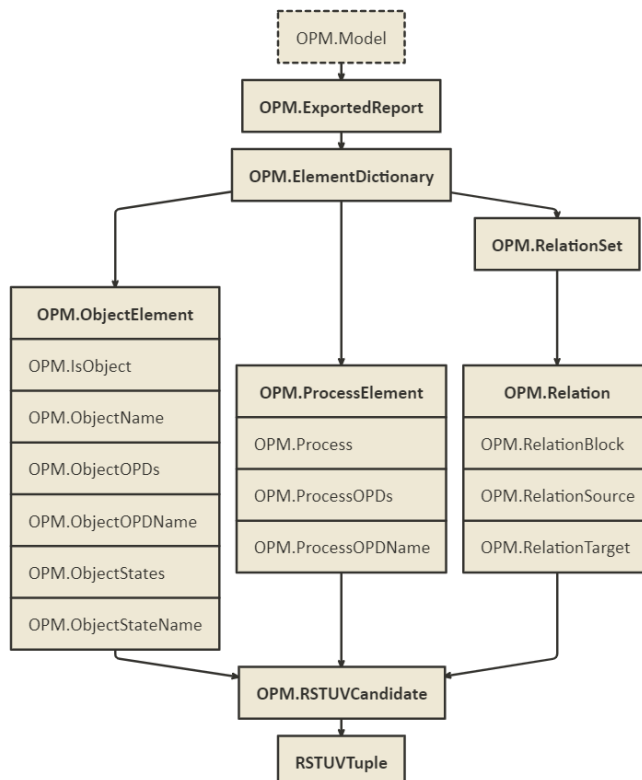


Figure 7. An ontology mapping functor schema from OPML to GDS RSTUV tuples extracts an *ElementDictionary* from the model's *ExportedReport*, and transforms *ElementDictionary* into three sets: *ObjectElement*, *ProcessElement*, and *RelationSet*. *RelationSet* is decomposed into a set of *Relations*. The functor computes attributes for each entity type from the report. For example, it lists the *ObjectOPDs* in which each *Object* is defined, and extracts the *Source* object/ process and the *Target* object/process for each *Relation*. Each entity is then transformed into a set of *RSTUVCandidates*. Duplicate candidates are removed before the set of *RSTUVTuple* is compiled. Illustration created using the www.NomnoML.com tool, a candidate for future MGVS transformation.

From `ExportedReport` we generate two interim categories: `ElementDictionary` and `OPLSpec`. We refer to both objects as sets, but due to the complicated and heterogenous structure of set members, we need further decomposition and transformation of set members to their appropriate data structures. For example, we need to find the OPDs in which each Object and Process appear.

`ElementDictionary` does not specify appearance of relations in OPDs, but the statements referring to the relations are specified in the OPL text accompanying each OPD. To extract this information, we need to analyze `OPLSpec`, identify the sentence that specifies each relation under the diagrams in which it is shown, and match the `OPD→OPLStatement` relation to the right Relation.

`OPLStatements` are compound, i.e., they aggregate multiple target entities having the same relation with a source entity into one `OPLStatement`. In fact, the OPL is a textual representation of the model that provides some information about the model. For example, the sentence “**Lane Keeping System** exhibits **Lane Keeping**, as well as **Improved Safety**” refers to two separate Exhibition relations shown in **Figure 2: Lane Keeping System → Lane Keeping, Lane Keeping System → Improved Safety**. This mandates additional text parsing, including the following steps:

1. Identify the position of the keyword “ exhibits ” in the `OPLStatement`;
2. Parse the left part of the sentence before the keyword position to extract the Source;
3. Parse the right part of the sentence after the keyword end to obtain a list of Targets;
4. Extract target entities from Targets by detecting the separators “ , ” and “ , ”, as well as “ ; ”;
5. Create a mapping {R=Exhibition, S=Source, T=Targets(t), U=uniqueID} for each t in Targets;

For each exhibition link in `ElementDictionary`, if the RST sub-tuple is found in the OPD, create a mapping {R=Inclusion, S=OPD, T=U of the relation}.

`OntologyMapping` is a generic transformation function that converts or extends ontological terms and execute all parts of the functor illustrated in **Figure 7**. A prototype of `OntologyMapping` has been implemented using MATLAB 2019b, with the capability to read a set of mappings for any ML and apply them to an input source that represents a model in that ML. The prototype supports reading model representations from various input sources, and storing the results to a Microsoft SQL Server database, Microsoft Excel file, text file, etc.

`OntologyMapping` starts from the raw input representing external models and other sources of information (e.g., `OPM.ExportedReport`). It continues **recursively** to either create new items or extend existing items. The `ExportedReport→ElementDictionary` transform, followed by the three transforms `ElementDictionary→{ObjectElement, ProcessElement, RelationSet}` create new items sets. `RelationSet` is further transformed into a new set of `Relation` items. This additional step is due to `OPCloud`’s exported report clustering of all relations of the same type in groups.

`OntologyMapping` extends `ObjectElement` with attributes such as `IsObject`, `ObjectName`, and `ObjectStates`. It similarly extends `ProcessElement` with the attributes `IsProcess`, `ProcessOPDs` etc., and `RelationSet` with the attributes `RelationBlock`, `RelationSource`, and `RelationTarget`.

The strength and robustness of `OntologyMapping` allows it to recursively search for additional mappings needed due to the creation of new items, but it would first make sure all the extended attributes are computed, because those attributes might be needed for creating new items. For instance, to create `RSTUVCandidates` that cover all the relevant relations pertaining to the `Process` item, we need to know which OPDs the process is in, and create a separate `RSTUVCandidate` with `R=Inclusion`, `S=ProcessElement(p).ProcessOPD`, and `T= ProcessElement(p).Process`, where p is a member of the `ProcessElement` set, `p=1,...,|ProcessElement|`.

`OntologyMapping` can receive an extensible data structure in the JavaScript Object Notation (JSON) that defines required conversions for any item. An example of the JSON extensible configuration for mapping `ProcessElement` to `RSTUVCandidate` is shown in **Figure 8**. The JSON object can define:

- Identity Attributes (`identityvars`): attributes whose values are converted to the Identity relation and the Classification relation.
 - The Identity relation maps each entry to a universally unique ID (UUID).

- The Classification relation maps each entry to its type, which is the attribute name. For example, each item in the `OPM.Diagram` column is mapped to an `RSTUVTuple` with `R=Classification`, `S=OPM.Diagram`, and `T=Item.UUID`.
- Pairs of Source and Target attributes with a specified relation (`sourceTargetPairs`) : pairs of attributes, for which each pair of attribute values is transformed into an RST tuple with a specific defined relation. For example, the `OPM.Object` and `OPM.ObjectState` attributes are mapped to a set of RST tuples with `R=StateSpecification`, `S={items in OPM.Object – one object per object definition block}`, and `T={items in OPM.ObjectState – one or more names of states}`
- Triplets of Source, Target, and Relation (`relationSourceTargetTriplets`) : triplets of attributes that constitute RST tuples that can be defined as RSTUV tuples. For example: under the `OPM.Relation` block, `R=Relation`, `S={set of all sources – basically just one object, process, or state}`, `T={set of all targets – basically just one object, process, or state}`.

The mapping function combinatorically searches for all the valid pair and triplet permutations and create a list of unique RSTUV candidates. Subsets of items in all relevant attributes that count only one item result in a single entry.

```
{
  "keyVar": "OPM.ProcessElement",
  "outputMode": 4,
  "identityVars": [
    "OPM.Diagram",
    "OPM.Process",
    "ModelView",
    "ModelID:Model"
  ],
  "sourceTargetPairs": [
    ["OPM.Inclusion", "ModelID", "ModelView"],
    ["OPM.Inclusion", "ModelView", "OPM.Process"],
    ["OPM.Inclusion", "ModelView", "OPM.Diagram"],
    ["OPM.Inclusion", "OPM.Diagram", "OPM.Process"]
  ]
}
```

Figure 8. A JSON structure defining the required transformations of `ProcessElement` attributes to `RSTUVCandidate` items. This example does not include `relationSourceTargetTriplets`.

An `RSTUVCandidate` referring to the same relation, source, and target entities might be created from multiple blocks. For example, since we derive identity relations for `OPM.Diagram` from both `ObjectElement` and `ProcessElement`, each OPD will have two `RSTUVCandidates`: one due to its inclusion of an object, and one due to its inclusion of a process. This is because diagram names are not defined separately in `ElementDictionary`, only indirectly through the listing of OPDs in which each object or process is visualized. We should converge all similar `RSTUVCandidates` with the same RST and assign the unique instances a unique id (U) and a version indicator (V). This also provides the capability to track changes in model versions through our platform.

`OntologyMapping` can store the results at any stage throughout the process. We can store the results in a file in various formats: Microsoft Excel (.xlsx), Comma-Separated Values (.csv), JSON (.json), or XML (.xml). We can also store the results in a relational database, such as SQL Server, through DDL queries, or as JSON/XML objects. Result Persistence is usually kept only for the endgame results, but retaining byproduct results is possible.

3.4 Transforming Graphs to Views and DSMs

The next step in the process is to define informative views on the model's GDS (set of RSTUV tuples), such as the DSMs discussed in section 2.3. As shown in **Figure 3** the essential graph, RST tuple set, and adjacency matrix are equivalent representations. Therefore, any relational pattern in the model, captured in the GDS, can be represented in a matrix. Furthermore, if the mapping via a selected relation or subset of relations makes sense or helps in reasoning about a problem, then we will be able to support this reasoning process by delivering a decision-supporting visualization in the form of a matrix based on the GDS based on the original model.

A DSM captures a mapping of row items to column items. We can specify a subset of relations and get a mapping of row items to column items according to the selected relations. We use SQL as a robust querying language, to reconstruct threads and spans of relations over the GDS, with the RSTUV tuples stored in a relational SQL Server database. Sequential SQL join queries make up

functors from the raw GDS to matrix or tensor datasets [72], which visualization and analysis tools can render as matrices. A common spreadsheet tool like Microsoft Excel can import the query results from the database management system, construct a pivot table on top of the raw data, and render the appropriate visualization. Pivot tables have a flexible structure that allows the analyst to adjust, transpose, and organize the matrix hierarchically for various visualization needs. Readily available data analysis tools like pivot tables simplify the analysis and minimize tool dependency. Additional visualization tools and techniques can be explored in the future.

Additional analysis can include tallies, sums, minima, maxima, subtotals on indications, or associated/converted matrix cells values. For example, if the number of allowed relations in each intersection must positive, we can quickly find the discrepancies (both visually and computationally).

3.5 Transforming Views into Concepts, and Concepts back into Models

The CMGVC loop's cognitive segment, $V \rightarrow C \rightarrow M$ consists of reasoning, decision-making, and action-taking. During this process, we study the views, mentally fuse the information with knowledge and beliefs, and create or revise a mental model, or concept. Based on the concepts we have in mind, we make decisions that alter the concept, or alter reality to match the revised concept. These cognitive view-to-concept and concept-to-model mappings close the loop by returning to the category of conceptual systems, *SYS*. A model of an existing system explains hypotheses, theories, or facts about the system. A model of a future system serves to inform system that will suit their concept. Stakeholders may later take action to execute decisions and adjust the model developers about the system's expected structure and behavior to fulfill the concept that the system was meant to realize. The conceptual transformation is a cognitive process that closes the loop and allows us to re-iterate through conceptual models of described or prescribed systems. Rigorous formulation of these transforms requires substantial grounding in cognitive psychology, and is beyond our current scope.

4. Assessment

The CMGVC cycle offers a significant departure from direct generation of views on models, and a more robust alternative to indirect generation of views based on DSML-specific representations.

In section 1 we have defined five propositions for this research:

- A. $M \rightarrow G \rightarrow V$ (a GDS-mediated transformation from model to view) is superior to $M \rightarrow V$ (a direct transformation from model to view)
- B. $M \rightarrow G \rightarrow V$ is superior to $M \rightarrow R \rightarrow V$ (a language-bound representation-mediated transformation from model to view)
- C. $M \rightarrow G$ is a feasible and valid transformation.
- D. $G \rightarrow V$ is a feasible and valid transformation.
- E. $G \rightarrow \text{DSM}$ is a feasible and valid transformation.

In sections 3 we have presented a categorical framework for converting conceptual system models from one ML, OPM, into a GDS, and deriving DSM views from the GDS. We have thus managed to validate Propositions C, D, and E regarding the existence of the building blocks of a composed transform $M \rightarrow G \rightarrow V$ based on $GV \circ MG$, where $MG: M \rightarrow G$, $GV: G \rightarrow V$.

To prove Propositions A and B we must compare the benefits and limitations of the CMGVC approach vis-à-vis the two alternative approaches:

- MV: Direct generation of views from a model, that we denote $V(M)$ or MV
- MRV: Indirect generation of views from a model via a common DSML-specific representation, that we would denote $V(R_{\text{DSML}}(M_{\text{DSML}}))$ or MRV

We define four lower-is-better (LIB) criteria for comparison, which reflect stakeholders needs:

- **C1: the number of required transformations** of M models to V views. For MG, MRV: sum of model-to-graph transformations (M) and graph-to-view transformations (V); For MV: product of models by views ($M \cdot V$).
- **C2: effort of creating new views for existing models**. For MG, MRV: one effort unit per view (graph-to-view or representation-to-view); For M: M effort units (models-to-view).

- **C3: effort of creating existing views for new models:** For MGV: a single effort unit (model-to-graph); for MV: V effort units (model-to-views); for MRV: V+1 effort units (model-to-representation and representation-to-views)
- **C4: dependency on DSML updates:** for MGV, MRV: a single effort unit for updates (either model-to-graph or model-to-representation); for MV: V effort units (model-to-views)

Since all criteria are LIB, the total score is also LIB. **Table 3** defines metrics for each criterion. Total scores for MGV, MV, and MRV are defined in Eq. (1), (2), and (3).

Table 3. Metrics for comparing approaches to generate views from models.

Criterion	Weight	MGV	MV	MRV
C1	W1	M+V	MV	M+V
C2	W2	1	M	1
C3	W3	1	V	V+1
C4	W4	1	V	1
Total	1	Eq. (1)	Eq. (2)	Eq. (3)

$$S_{MGV} = W_1(M + V) + W_2 + W_3 + W_4 \quad (1)$$

$$S_{MV} = W_1(MV) + W_2M + W_3V + W_4V \quad (2)$$

$$S_{MRV} = W_1(M + V) + W_2M + W_3(V + 1) + W_4 \quad (3)$$

MGV is superior to MRV (Proposition B) if $S_{MGV} \leq S_{MRV}$: $W_1(M + V) + W_2 + W_3 + W_4 \leq W_1(M + V) + W_2M + W_3(V + 1) + W_4$. Cancelling equal terms, we obtain $W_2 + W_3 \leq W_2M + W_3(V + 1)$. Homogenizing we obtain $0 \leq W_2(M - 1) + W_3(V)$. This inequality always holds, and therefore $S_{MGV} \leq S_{MRV}$, and MGV is superior to any MRV with a DSML-specific representation:

$$S_{MGV} \leq S_{MRV} \Leftrightarrow \mathbf{MGV} \succcurlyeq \mathbf{MRV} \quad (4)$$

Conditions for MGV superiority to MV: $S_{MGV} \leq S_{MV} \Leftrightarrow W_1(M + V) + W_2 + W_3 + W_4 \leq W_1(MV) + W_2M + W_3V + W_4V$. Homogenizing we obtain the following:

$$S_{MGV} \leq S_{MV} \Leftrightarrow W_1(MV - M - V) + W_2(M - 1) + W_3(V - 1) + W_4(V - 1) \geq 0 \quad (5)$$

We try Eq. (5) with combinations of V=1, V=2, V>2 with M=1, M=2, M>2, as summarized in **Table 4**. For all practical purposes, multiple MLs $M \geq 2$, and at least 2 views (we already have 3),

$$M \geq 2, V \geq 2 \Leftrightarrow S_{MGV} \leq S_{MV} \Leftrightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV} \quad (6)$$

Table 4. Preference Relations of MGV over MV depending on the number of MLs (M) and views (V).

MV is preferred when M=V=1; MGV is preferred when $M \geq 2$ & $V \geq 2$. When $M = 1$ & $V \geq 2$ or $M \geq 2$ & $V = 1$ preference rotates based on weighting. MGV is preferable when weights are equal.

	V=1	V=2	V>2
M=1	$\nexists W_1(-1) \geq 0$ $\Rightarrow S_{MV} \leq S_{MGV} \Rightarrow \mathbf{MGV} \preccurlyeq \mathbf{MV}$	$W_3 + W_4 \leq W_1$ $\Rightarrow S_{MV} \leq S_{MGV} \Rightarrow \mathbf{MGV} \preccurlyeq \mathbf{MV}$	$(W_3 + W_4) \leq \frac{W_1}{3}$ $\Rightarrow S_{MV} \leq S_{MGV} \Rightarrow \mathbf{MGV} \preccurlyeq \mathbf{MV}$
		$W_3 + W_4 \geq W_1$ $\Rightarrow S_{MGV} \leq S_{MV} \Rightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV}$	$(W_3 + W_4) \geq \frac{W_1}{3}$ $\Rightarrow S_{MGV} \leq S_{MV} \Rightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV}$
M=2	$W_1 \geq W_2$ $\Rightarrow S_{MV} \leq S_{MGV} \Rightarrow \mathbf{MGV} \preccurlyeq \mathbf{MV}$	$W_2 + W_3 + W_4 \geq 0$ $\Rightarrow S_{MGV} \leq S_{MV} \Rightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV}$	$mW_1 + W_2 + n(W_3 + W_4) \geq 0$ $m, n \in \mathbb{N}$ $\Rightarrow S_{MGV} \leq S_{MV} \Rightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV}$
	$W_2 \geq W_1$ $\Rightarrow S_{MGV} \leq S_{MV} \Rightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV}$		
M>2	$W_2 \leq \frac{W_1}{M-1}$ $\Rightarrow S_{MV} \leq S_{MGV} \Rightarrow \mathbf{MGV} \preccurlyeq \mathbf{MV}$	$mW_1 + nW_2 + W_3 + W_4 \geq 0$ $m, n \in \mathbb{N}$ $\Rightarrow S_{MGV} \leq S_{MV} \Rightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV}$	$pW_1 + mW_2 + n(W_3 + W_4) \geq 0$ $m, n, p \in \mathbb{N}$ $\Rightarrow S_{MGV} \leq S_{MV} \Rightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV}$
	$W_2 \geq \frac{W_1}{M-1}$ $\Rightarrow S_{MGV} \leq S_{MV} \Rightarrow \mathbf{MGV} \succcurlyeq \mathbf{MV}$		

With this analysis, we have managed to corroborate Proposition A and Proposition B.

5. Application

5.1 Representation and Analysis of Process-to-Process Input-Output Exchange

Table 1 illustrates two examples of SAMs that map various system architecture aspects to each other. Both SAMs include a mapping of processes to each other, with **Table 1-a** being more explicit about the mapping of process to process via operands. The operands serve as indirect process mediators: the output of one process is the input of another. However, there is no explicit mapping in the model in the form $\langle P, O, P \rangle$. Therefore, this mapping must be composed. This composition is defined in the following manner, using SQL queries to retrieve and cross the data from the raw set of RSTUV tuples:

1. Find all the processes, i.e., targets in tuples where the source is 'OPM.Process' and the relation is 'Classification'.
2. Find all outputs, i.e., targets in tuples where the relation is 'Result', and retrieve the source process item.
3. Find all tuples in which the above outputs are sources in a Consumption relation (i.e., inputs), and retrieve the target process items.
4. Cross the process-output set with the input-process set such that output = input.
5. Cross the process set (1) with the output-generating process set (2) while keeping all processes including those that are generating no output.
6. Cross the process set (1) with the input-receiving process set (3) while keeping all processes including those that are receiving no input.
7. Layout a matrix with output-generating processes as rows, input-receiving processes as columns, and identity of matching output-input (*onput* [68]) item as the cell value.

Figure 9 illustrates the MGV transform into three views of a generic OPM model (**Figure 9-a**) with seven processes and eight objects that the processes generate or consume:

8. **Figure 9-b**: A visual graph with onputs on nodes and processes on edges highlights the systemic morphisms of onputs through processes and is useful for showing onput transformations, e.g. matter to energy, energy to data, etc., as discussed in section 3.1. The SysML Block Diagram notation renders an object-on-node graph.
9. **Figure 9-c**: A visual graph with processes on nodes and onputs on edges – the dual graph of Figure 9-b – highlights process flow and is useful for showing functional compositions through shared onputs, process threads, and functional boundaries and interfaces. The SysML Activity Diagram notation renders a process-on-node graph.
10. **Figure 9-d**: A process-operand-process (POP) exchange matrix maps processes to each other via shared onputs they exchange, including indications for partial exchanges (inputs without source processes, processes without input or output, outputs without target processes, etc.), and facilitates visual or computational analysis, verification, and validation.

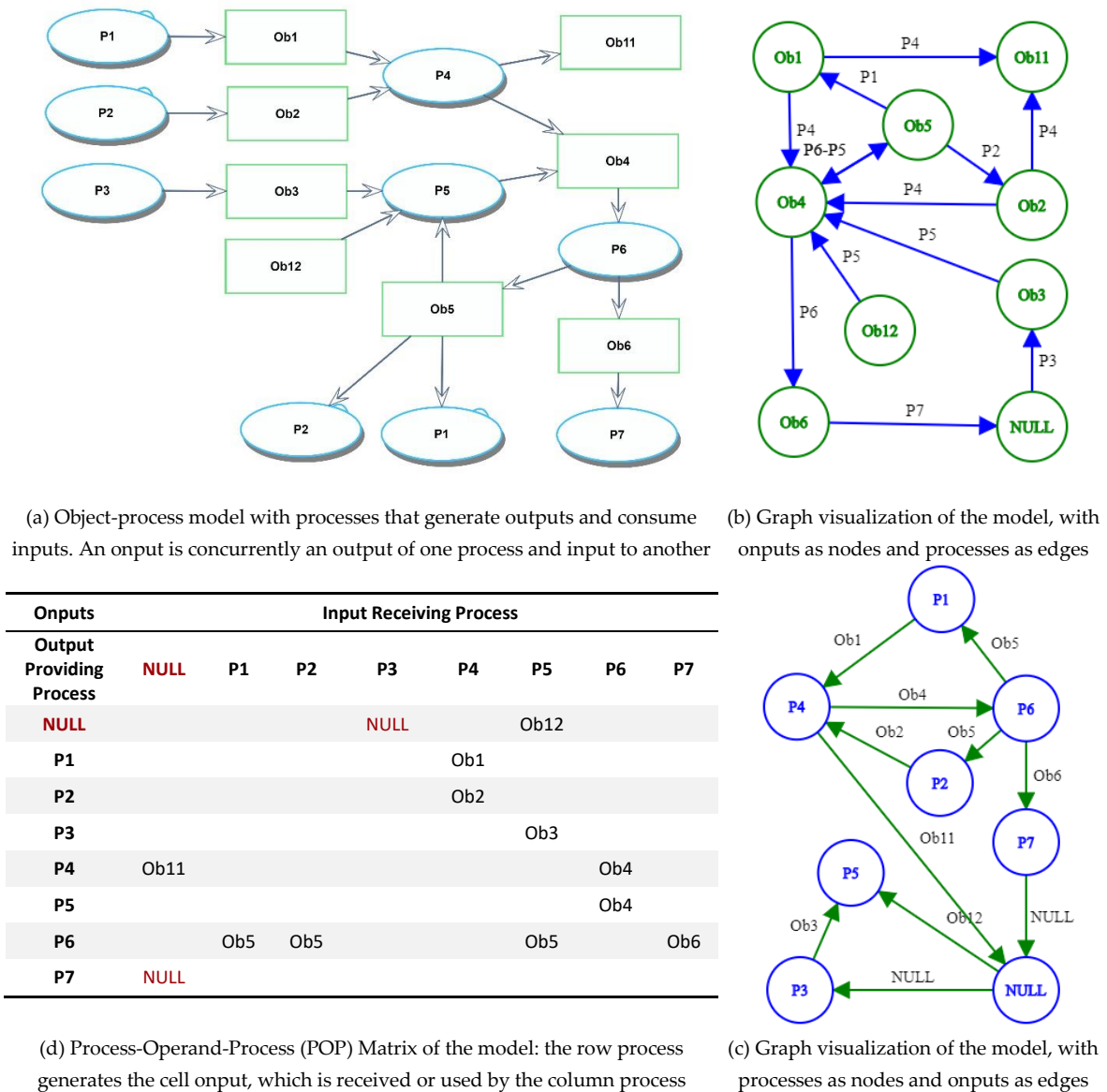


Figure 9. Transforming an OPM model with processes and onputs (a) through the MGV transform into three renditions: b) an input-on-node/process-on-edge graph, c) a process-on-node/onput-on-edge graph, and d) a process-to-process output exchange matrix, which is based on the same query as the process-on-node graph. NULL processes and onputs (in all renditions) serve as placeholders for missing or partial relations, e.g., an input without a source process (e.g. Ob12), a process without an input (e.g. P3) or without an output (e.g. P7), and an output without a target process (e.g. Ob11). Both graph visualizations were rendered using https://csacademy.com/app/graph_editor/.

5.2 The Lane Keeping System Revisited

Our lane keeping system running example can now be analyzed according to the CMGVC cycle. An in-zoomed diagram of the Lane Keeping System, specifying the functions performed by the system and their interactions, is illustrated in **Figure 10**. This is an intentionally partial diagram, with some critical things missing. As we study this diagram we should notice, for instance, that the Lane Crossing object has no source, and that the Road is consumed by the Imaging Road process, rather than remain an instrument as it was specified in the topmost diagram. It is difficult to ensure continuity and consistency this way, and the problem intensifies as the model grows bigger with more objects, processes, and diagrams.

Our lane keeping system running example can now be analyzed according to the CMGVC cycle. An in-zoomed diagram of the Lane Keeping System, specifying the functions performed by the

system and their interactions, is illustrated in **Figure 10**. This is an intentionally partial diagram, with some critical things missing. As we study this diagram we should notice, for instance, that the Lane Crossing object has no source, and that the Road is consumed by the Imaging Road process, rather than remain an instrument as it was specified in the topmost diagram. It is difficult to ensure continuity and consistency this way, and the problem intensifies as the model grows bigger with more objects, processes, and diagrams.

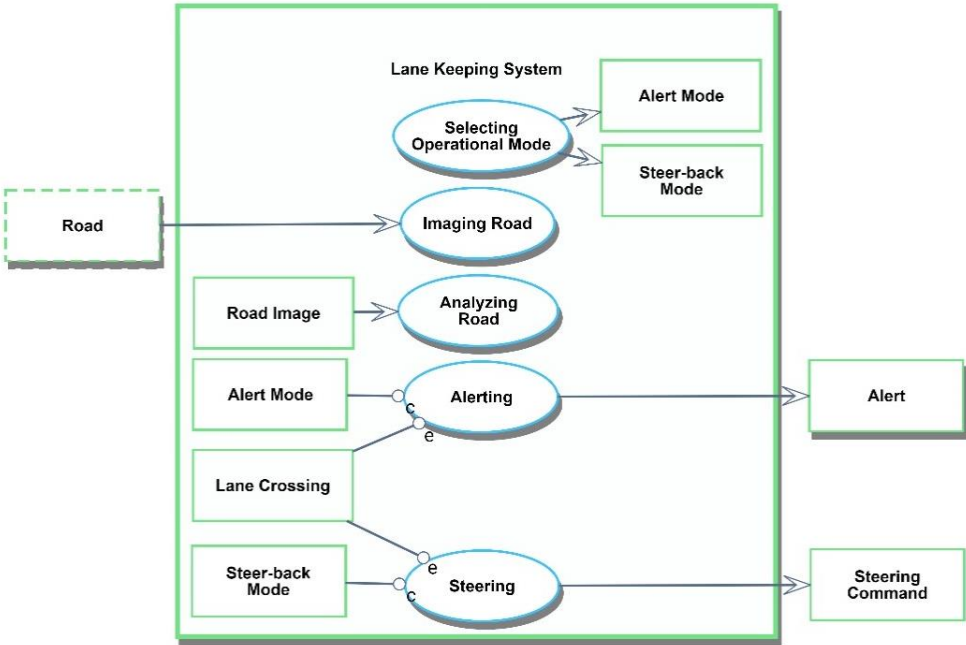


Figure 10. Lane Keeping System zooms into Alert Mode, Steer-back Mode, Road Image, Alert Mode, Lane Crossing, and Steer-back Mode, in that vertical sequence, as well as Alerting, Analyzing Road, Imaging Road, Selecting Operational Mode, and Steering (This caption is in fact a formal OPL sentence that specifies the in-zooming of Lane Keeping System in a new diagram).

Analyzing this model with a POP matrix could help detect some discrepancies in the model. **Table 5** is a POP matrix that was derived from the model through the same SQL query defined in section 5.1 for the POP matrix-supporting records, and rendered as a matrix using a MS Excel pivot table on the query results. As the Lane Keeping System model utilizes additional procedural relations besides the result and consumption: the instrument relation is used for non-consumable resources and the invocation relation is used for specifying process activation by another process, implying an unspecified interface object. Additional metrics were added to the POP matrix, including the number of inputs and number of outputs per process (calculated on the columns and rows, respectively).

The POP matrix highlights several issues in the model, which could be helpful for reviews by various stakeholders – customers, system architects, process or function owners, etc. We summarize proposed corrections by entity and issue in **Table 6**. It would be easy to apply these to the model and reiterate to obtain the revised POP matrix. As more details are added to the model, the POP matrix is likely to grow and detect new issues, in a continuous concept revision and improvement cycle.

Table 5. The POP matrix of the Lane Keeping System model maps the modeled processes to each other via output exchange, instrument providing, and invocation. The matrix uses color coding to quickly highlight discrepancies and potential issue. Red cells with a value of 0 indicate processes without inputs or outputs, while lightly-colored cells indicate that further attention is needed.

Target Process Source Process	(Null)	Alerting	Analyzing Road	Imaging Road	Lane Keeping	Selecting Operational Mode	Steering	Driving	Propelling	Outputs per Process
(Null)		Lane Crossing	Road Image	Road	Road		Lane Crossing		Road	6
Lane Keeping	Improved Safety							Alert	Steering Command	3
Selecting Operational Mode		Alert Mode					Steer-back Mode			2
Imaging Road										0
Analyzing Road										0
Alerting								Alert		1
Driving									Invocation	1
Steering									Steering Command	1
Propelling								Invocation		1
Inputs per Process	1	2	1	1	1	0	2	3	4	

Table 6. Issues identified in Lane Keeping System model POP matrix and possible corrections in the model. Critical issues are marked in red.

Entity	Issue	Correction
Lane Keeping	no functional decomposition	Unfold Lane Keeping in a separate diagram and link it to the internal functions in Lane Keeping System to create functional decomposition of top level system function to lower-level functions
Selecting Operational Mode	no input	Add input from Driver
Imaging Road	no output	Link through a Result relation to Road Image
Analyzing Road	no output	Link through a Result relation to Lane Crossing Specify as attribute of Lane Keeping System .
Improved Safety	no receiver	Since this object represents emergent value, it is not recommended to associate it with a specific function
Road Image	no provider	Link Imaging Road through a Result relation
Lane Crossing	no provider	Link Analyzing Road through a Result relation
Road	no provider	Add Effect Link from Propelling, indicating that vehicle movement affects the immediate portion of interest of the road with which the vehicle interacts
Driving invokes Propelling	no explicit output	Consider replacing invocation by driver command gestures (turning, signaling, braking, etc.)
Propelling invokes Driving	no explicit output	Consider replacing invocation by vehicle response to driver action

6. Conclusion

Systems engineering is undergoing a digital transformation, compounded with the constant growth of systems complexity and interconnectedness. Conceptual modeling is becoming more common for representing complex systems, and more critical for generating deliverables that impact the digital value chain [10]. Considering the challenge and opportunity associated with these trends, we set out to find robust foundations for MBSE interoperability and collaboration across digital enterprises.

We have explored the potential of Category Theory to serve as an underlying formalism for systems engineering, particularly in the context of MBSE. Category Theory is an appropriate holistic paradigm, a state of mind, and a formal foundation for the model transformation and reasoning pipeline, which is essential for a smooth, rational, and reliable MBSE cycle that constantly improves, corrects, and refines system architecture specifications.

Drawing on the Curry-Howard-Lambek Correspondence of programming languages and categories, we show that modeling languages, like programming languages, are valid categories. We have used categorical structures to map system concepts to model concept, models to robust graph data structures (GDS), and GDS to decision-supporting views – graphs and matrices.

The Concept-Model-Graph-View-Concept (CMGVC) cycle facilitates the transformation of conceptual models to decision-supporting views. We have shown that it would be imperative to include an intermediate common representation in the form of a GDS, which serves as a common outlet to all MLs, and a common basis for all views, visualizations, and reports. We have proven the superiority of our approach to direct mappings and ML-bound representation-based mappings.

The CMGVC approach has several advantages:

- Using the CMGVC approach, **stakeholders and decision-makers will be able to derive critical information and insight** regarding system development and operation from the system model, rather than through a disparate information gathering and presentation channel, which is the common practice nowadays.
- The preferential dominance that we have proven in section 4 facilitates **efficiency in model analytics**, and thus encourages further adoption.
- The transition through GDS **enhances system understanding by adding another modality: graphs**, which map concepts and relations through one common substantial representation.
- The simple-yet-robust GDS can be a prime facilitator of **MBSE interoperability and collaboration** across digital value chains.
- Subject matter experts will be able to leverage the CMGVC approach via semantic and ontological frameworks to **better represent emerging patterns and concepts**.

We have demonstrated how a model can be represented in multiple ways that are all based on the same GDS representation of that model. These include two visual graph renditions, which are also duals: a process-on-node/output-on-edge graph and an output-on-node/process-on-edge graph and a process-to-process operand exchange matrix.

Both dual graph renditions have uses and advantages. Being able to generate both graphs from the same data structure is an important benefit, which also ensures consistency and complementarity. The SysML Internal Block Diagram (IBD) uses an object-on-node graph, while the SysML Activity Diagram uses a process-on-node graph, and the State Chart uses a state-on-node graph. These three aspects can be generated from the same GDS representation of an OPM model. Therefore, the GDS can facilitate modeling language translation. We intend to explore this direction in future studies.

The process-operand-process exchange matrix is an important tool for architectural analysis, and it captures a non-trivial compositional mapping relation that we could build by composing relation segments in the GDS through robust data retrieval and integration queries. Obtaining and maintaining such a matrix manually is a significant cognitive effort. The ability to dynamically visualize the matrix using a pivot table is another strong benefit, due to both its accessibility to many potential adopters, and its ability to accommodate their evolving needs.

Further exploring the applicability of this approach to various MLs, we plan to demonstrate that the CMGVC approach can be robustly suited for various other MLs and views. Particularly, we plan to explore the transformation of SysML and Simulink models via the CMGVC pipeline and to create

meaningful source-agnostic views, including DSMs, sub-graphs, and specifications in other MLs. Isomorphic mapping capabilities out of and back into the same ML are also a major challenge.

Reconciling the MBSE paradigm with the Discrete Event Simulation (DEVS) paradigm is a recent ongoing effort [73]. Ongoing collaboration between the MBSE and DEVS communities has been institutionalized in a joint workgroup for the International Council on Systems Engineering (INCOSE) and International Association for Engineering Modeling and Simulation (NAFEMS) [74]. Our proposed framework can facilitate interoperability across MBSE and DEVS platforms, tools, and models. It alleviates efforts to find and apply an interoperability standard that would appeal to both MBSE and DEVS. The reason is that graph-theoretic DEVS algorithms are abundant, and graph-representable problems have been studied with DEVS technology and methodology for decades, for example as part of the Agent-Based Modeling & Simulation (ABMS) paradigm [75, 76]. This direction should be explored as part of future research.

One limitation in our approach is its scalability. We plan to build a service-oriented software platform that will provide the robust transformation capabilities as web services to MBSE practitioners and researchers, who will be able to upload or plug-in live models and benefit from ad-hoc or continuous transformations and visualizations. MBSE experts should also be able to define new mappings, maintain and enhance existing mappings, and define sets of views to run on specific models. These will provide continuous visualization or integration with external systems across the digital enterprise. We plan to employ state-of-the-art information visualization techniques to provide stakeholders with flexible, dynamic, and elegant decision-supporting views.

Author Contributions: Conceptualization, Y.M., E.C., J.F.; methodology, Y.M, J.F., E.C.; software, Y.M.; validation, Y.M, J.F.; formal analysis, Y.M.; writing, Y.M., J.F.; visualization, Y.M., J.F.; supervision, E.C.; funding acquisition, Y.M, J.F., E.C. All authors have read and agreed to the published version of the manuscript.

Funding: Y.M.'s research was conducted as part of a research fellowship, funded by the MIT-Technion Post-Doctoral Research Fellowship Program. J.F.'s research was funded by DARPA under agreements HR00112090067 and W911NF2010292.

Acknowledgments: We thank David Spivak from MIT's Department of Mathematics for planting the seeds for the collaboration that resulted in this research, for inspiring it, and for very useful comments. We also thank Prof. Dov Dori from Technion and Prof. Yoram Reich from Tel-Aviv University for their useful comments.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in designing the study; collecting, analysing, or interpreting data; writing the manuscript, or in deciding to publish the results.

7. References

1. Partridge, C., Gonzalez-Perez, C., Henderson-Sellers, B.: Are conceptual models concept models? *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*. 8217 LNCS, 96–105 (2013). https://doi.org/10.1007/978-3-642-41924-9_9.
2. Bailer-Jones, D.M.: Scientific models in philosophy of science. (2009). <https://doi.org/10.1080/02698595.2010.543353>.
3. Soderborg, N.R., Crawley, E.F., Dori, D.: System Function and Architecture: OPM-Based Definitions and Operational Templates. *Commun. ACM*. 46, 67–72 (2003). <https://doi.org/10.1145/944217.944241>.
4. Dori, D.: *Model-Based Systems Engineering with OPM and SysML*. Springer, New York (2016). <https://doi.org/10.1007/978-1-4939-3295-5>.
5. McDermott, T.A., Hutchinson, N., Clifford, M., Van Aken, E., Slado, A., Henderson, K.: Benchmarking the Benefits and Current Maturity of Model-Based Systems Engineering across the Enterprise. *Systems Engineering Research Center (SERC)* (2020).
6. Bondar, S., Hsu, J.C., Pfouga, A., Stjepandić, J.: Agile digital transformation of System-of-Systems architecture models using Zachman framework. *J. Ind. Inf. Integr.* 7, 33–43 (2017). <https://doi.org/10.1016/j.jii.2017.03.001>.

7. Hale, J.P., Zimmerman, P., Kukkala, G., Guerrero, J., Kobryn, P., Puchek, B., Bisconti, M., Baldwin, C., Mulpuri, M.: Digital Model-based Engineering: Expectations, Prerequisites, and Challenges of Infusion. NASA (2017).
8. Bone, M., Blackburn, M., Kruse, B., Dzielski, J., Hagedorn, T., Grosse, I.: Toward an Interoperability and Integration Framework to Enable Digital Thread. *Systems*. 6, 46 (2018). <https://doi.org/10.3390/systems6040046>.
9. Hagedorn, T., Bone, M., Kruse, B., Grosse, I., Blackburn, M.: Knowledge Representation with Ontologies and Semantic Web Technologies to Promote Augmented and Artificial Intelligence in Systems Engineering. *Insight*. 23, 15–20 (2020). <https://doi.org/10.1002/inst.12279>.
10. Mordecai, Y., de Weck, O.L., Crawley, E.F.: Towards an Enterprise Architecture for a Digital Systems Engineering Ecosystem. *Conf. Syst. Eng. Res.* (2020).
11. Subrahmanian, E., Levy, S.N., Westerberg, A.W., Monarch, I., Konda, S.L., Reich, Y.: Equations aren't enough: Informal modeling in design. *Artif. Intell. Eng. Des. Anal. Manuf.* 7, 257–274 (1993). <https://doi.org/10.1017/S0890060400000354>.
12. Sharpe, J.E.E., Bracewell, R.H.: Application of bond graph methodology to concurrent conceptual design of interdisciplinary systems. In: *IEEE Systems Man and Cybernetics Conference - SMC* (1993). <https://doi.org/https://doi.org/10.1109/ICSMC.1993.384711>.
13. Ernadote, D.: An ontology mindset for system engineering. *1st IEEE Int. Symp. Syst. Eng. ISSE 2015 - Proc.* 454–460 (2015). <https://doi.org/10.1109/SysEng.2015.7302797>.
14. Cotter, M., Hadjimichael, M., Markina-khusid, A., York, B.: Automated Detection of Architecture Patterns in MBSE Models. In: *Conference on Systems Engineering Research (CSER)* (2020).
15. Reich, Y., Konda, S., Subrahmanian, E., Cunningham, D., Dutoit, A., Patrick, R., Thomas, M., Westerberg, A.W.: Building agility for developing agile design information systems. *Res. Eng. Des. - Theory, Appl. Concurr. Eng.* 11, 67–83 (1999). <https://doi.org/10.1007/PL00003884>.
16. United States Department of Defense (DoD): The DoDAF Architecture Framework Version 2.02, <https://dodcio.defense.gov/Library/DoD-Architecture-Framework/>, last accessed 2020/12/03.
17. Fong, B., Spivak, D.I.: Seven Sketches in Compositionality: An Invitation to Applied Category Theory. (2018).
18. Chakraborty, S.: Curry-Howard-Lambek Correspondence. (2011).
19. Hamilton, M.: Category Theory and the Curry-Howard-Lambek Correspondence. (2016).
20. ISO/TC 184: ISO 19450 Automation systems and integration — Object-Process Methodology. International Organization for Standardization (ISO), Geneva, Switzerland (2015).
21. Crawley, E., Cameron, B., Selva, D.: *Systems Architecture: Strategy and Product Development for Complex Systems*. Prentice Hall (2015).
22. Mordecai, Y., Dori, D.: Model-Based Operational-Functional Unified Specification for Mission Systems. In: *10th Annual IEEE International Systems Conference (SysCon)*. IEEE, Orlando FL, USA (2016). <https://doi.org/10.1109/SYSCON.2016.7490662>.
23. Mordecai, Y., Dori, D.: Model-based requirements engineering: Architecting for system requirements with stakeholders in mind. In: *IEEE International Symposium on Systems Engineering, ISSE* (2017). <https://doi.org/10.1109/SysEng.2017.8088273>.
24. Osorio, C.A., Dori, D., Sussman, J.: COIM: An Object-Process Based Method for Analyzing Architectures of Complex, Interconnected, Large-Scale Socio-Technical Systems. *Syst. Eng.* 14, (2011). <https://doi.org/DOI 10.1002/sys.20185>.

25. Dori, D., Jbara, A., Levi, N., Wengrowicz, N.: Object-Process Methodology, OPM ISO 19450 – OPCLoud and the Evolution of OPM Modeling Tools, https://www.ppi-int.com/wp-content/uploads/2018/01/SyEN_61.pdf, (2018).
26. Levi-Soskin, N., Shaoul, R., Kohen, H., Jbara, A., Dori, D.: Model-Based Diagnosis with FTTell: Assessing the Potential for Pediatric Failure to Thrive (FTT) During the Perinatal Stage. In: Wrycza, S. and Maślankowski, J. (eds.) SIGSAND/PLAIS, LNBIP 359. pp. 37–47. Springer Nature Switzerland AG (2019). https://doi.org/10.1007/978-3-030-29608-7_4.
27. Dori, D., Kohen, H., Jbara, A., Wengrowicz, N., Lavi, R., Levi-Soskin, N., Bernstein, K., Shani, U.: OPCLoud: An OPM Integrated Conceptual-Executable Modeling Environment for Industry 4.0. In: Kenett, R.S., Swarz, R.S., and Zonnenshain, A. (eds.) Systems Engineering in the Fourth Industrial Revolution: Big Data, Novel Technologies, and Modern Systems Engineering. Wiley (2020).
28. Mordecai, Y., James, N.K., Crawley, E.F.: Object-Process Model-Based Operational Viewpoint Specification for Aerospace Architectures. In: IEEE Aerospace Conference. IEEE (2020). <https://doi.org/10.1109/AERO47225.2020.9172685>.
29. Ford Motor Company: Lane-Keeping System, <https://www.youtube.com/watch?v=8O3u20MBmsE>, last accessed 2020/11/20.
30. Dori, D.: ViSWeb - The Visual Semantic Web: Unifying human and machine knowledge representations with Object-Process Methodology. VLDB J. 13, 120–147 (2004). <https://doi.org/10.1007/s00778-004-0120-x>.
31. Shani, U., Jacobs, S., Wengrowicz, N., Dori, D.: Engaging ontologies to break MBSE tools boundaries through semantic mediation. In: Conference on Systems Engineering Research. INCOSE (2016).
32. Zhu, Y., Wan, J., Zhou, Z., Chen, L., Qiu, L., Zhang, W., Jiang, X., Yu, Y.: Triple-to-text: Converting RDF triples into high-quality natural languages via optimizing an inverse KL divergence. SIGIR 2019 - Proc. 42nd Int. ACM SIGIR Conf. Res. Dev. Inf. Retr. 1, 455–464 (2019). <https://doi.org/10.1145/3331184.3331232>.
33. Object Management Group: Unified Architecture Framework Profile (UAFP). (2019).
34. Browning, T.R.: Design Structure Matrix Extensions and Innovations: A Survey and New Opportunities. IEEE Trans. Eng. Manag. 63, 27–52 (2016). <https://doi.org/10.1109/TEM.2015.2491283>.
35. Sharon, A., Dori, D., de Weck, O.: Model-Based Design Structure Matrix: Deriving a DSM from an Object-Process Model. In: Second International Symposium on Engineering Systems (2009). <https://doi.org/10.1002/sys>.
36. Sharon, A., Dori, D.: A Project–Product Model–Based Approach to Planning Work Breakdown Structures of Complex System Projects. Systems. (2014). <https://doi.org/10.1109/JSYST.2013.2297491>.
37. Do, S., Weck, O. De: A Grammar for Encoding and Synthesizing Life Support. In: 44th International Conference on Environmental Systems (2014).
38. Wilschut, T., Etman, L.F.P., Rooda, J.E., Vogel, J.A.: Generation of a function-component-parameter multi-domain matrix from structured textual function specifications. Res. Eng. Des. 29, 531–546 (2018). <https://doi.org/10.1007/s00163-018-0284-9>.
39. Knippenberg, S.C.M., Etman, L.F.P., Wilschut, T., van de Mortel-Fronczak, J.A.: Specifying Process Activities for Multi-Domain Matrix Analysis Using a Structured Textual Format. Proc. Des. Soc. Int. Conf. Eng. Des. 1, 1613–1622 (2019). <https://doi.org/10.1017/dsi.2019.167>.
40. Object Management Group: OMG Systems Modeling Language (OMG SysML™) Version 1.4. Object Management Group (OMG) (2015). <https://doi.org/10.1002/j.2334-5837.2009.tb01046.x>.
41. Au-Yong-Oliveira, M., Moutinho, R., Ferreira, J.J.P., Ramos, A.L.: Present and future languages – How innovation has changed us. J. Technol. Manag. Innov. 10, 166–182 (2015). <https://doi.org/10.4067/S0718->

- 27242015000200012.
42. Johnson, J.M.: Analysis of Mission Effectiveness: Modern System Architectures Tools For Project Developers, (2017).
 43. Kong, P.O.: Spreadsheet-Based Graphical User Interface For Modeling Of Products Using The Systems Engineering Process, (2014).
 44. Thalheim, B.: The Theory of Conceptual Models, the Theory of Conceptual Modelling and Foundations of Conceptual Modelling. In: Embley, D.W. and Thalheim, B. (eds.) Handbook of Conceptual Modeling. pp. 543–577. Springer-Verlag Berlin Heidelberg (2011). <https://doi.org/10.1007/978-3-642-15865-0>.
 45. Wymore, W.A.: Model-Based Systems Engineering. CRC Press (2000).
 46. Breiner, S., Subrahmanian, E., Jones, A.: Categorical foundations for system engineering. *Discip. Conver. Syst. Eng. Res.* 449–463 (2017). <https://doi.org/10.1007/978-3-319-62217-0>.
 47. Subrahmanian, E., Reich, Y., Krishnan, S.: The Story of n-Dim. In: We Are Not Users: Dialogues, Diversity, and Design. pp. 181–196. MIT Press (2020). <https://doi.org/10.1525/9780520938342-009>.
 48. Breiner, S., Pollard, B., Subrahmanian, E.: Workshop on Applied Category Theory: Bridging Theory and Practice. (2020). <https://doi.org/https://doi.org/10.6028/NIST.SP.1249>.
 49. Koo, H.-Y.B.: A Meta-language for Systems Architecting, (2005).
 50. Koo, B., Hurd, A., Loda, D., Dori, D., Crawley, E.F.: Architecting Systems Under Uncertainty with Object-Process Networks. In: International Conference on Complex Systems (ICCS'04). , Boston, MA, USA (2004).
 51. Cafezeiro, I., Haeusler, E.H.: Semantic interoperability via category theory. 26th Int. Conf. Concept. Model. (2007).
 52. Spivak, D.I., Kent, R.E.: Ologs: A categorical framework for knowledge representation. *PLoS One.* 7, (2012). <https://doi.org/10.1371/journal.pone.0024274>.
 53. Mabrok, M.A., Ryan, M.J.: Category theory as a formal mathematical foundation for model-based systems engineering. *Appl. Math. Inf. Sci.* 11, 43–51 (2017). <https://doi.org/10.18576/amis/110106>.
 54. Luzeaux, D.: A Formal Foundation of Systems Engineering. In: F. Boulanger et al. (eds.) (ed.) Complex Systems Design & Management. Springer International Publishing Switzerland (2015). https://doi.org/10.1007/978-3-319-11617-4_10.
 55. Baez, J.C., Fong, B., Pollard, B.S.: A compositional framework for Markov processes. *J. Math. Phys.* 57, (2016). <https://doi.org/10.1063/1.4941578>.
 56. Censi, A.: Uncertainty in Monotone Codesign Problems. *IEEE Robot. Autom. Lett.* 2, 1556–1563 (2017). <https://doi.org/10.1109/LRA.2017.2674970>.
 57. Censi, A.: A Class of Co-Design Problems with Cyclic Constraints and Their Solution. *IEEE Robot. Autom. Lett.* 2, 96–103 (2017). <https://doi.org/10.1109/LRA.2016.2535127>.
 58. Kibret, N., Edmonson, W., Gebreyohannes, S.: Category theoretic based formalization of the verifiable design process. 13th Annu. IEEE Int. Syst. Conf. (2019). <https://doi.org/10.1109/SYSCON.2019.8836804>.
 59. Gebreyohannes, S., Edmonson, W., Esterline, A.: Formal behavioral requirements management. *IEEE Syst. J.* 12, 3006–3017 (2018). <https://doi.org/10.1109/JSYST.2017.2775740>.
 60. Halter, M., Herlihy, C., Fairbanks, J.: A Compositional Framework for Scientific Model Augmentation. (2019).
 61. Herlihy, C., Cao, K., Reparti, S., Briscoe, E., Fairbanks, J.: Semantic Program Analysis for Scientific Model Augmentation. In: Modeling the World's Systems (2019).
 62. Breiner, S., Sriram, R.D., Subrahmanian, E.: Compositional Models for Complex Systems. (2019). <https://doi.org/10.1016/b978-0-12-817636-8.00013-2>.

63. Kovalyov, S.P.: Leveraging category theory in model based enterprise. *Adv. Syst. Sci. Appl.* 20, 50–65 (2020). <https://doi.org/10.25728/assa.2020.20.1.781>.
64. Libkind, S.: *An Algebra of Resource Sharing Machines*. (2020).
65. Reich, Y., Konda, S.L., Levy, S.N., Monarch, I.A., Subrahmanian, E.: New roles for machine learning in design. *Artif. Intell. Eng.* 8, 165–181 (1993). [https://doi.org/10.1016/0954-1810\(93\)90003-X](https://doi.org/10.1016/0954-1810(93)90003-X).
66. Needham, M., Hodler, A.E.: *Graph Algorithms - Practical Examples in Apache Spark & Neo4j*. O'Reilly Media, Inc. (2019).
67. Patterson, E.: Hausdorff and Wasserstein metrics on graphs and other structured data. In: *American Mathematical Society Sectional Meeting on Category Theory* (2019).
68. Mordecai, Y., Orhof, O., Dori, D.: Model-Based Interoperability Engineering in Systems-of-Systems and Civil Aviation. *IEEE Trans. Syst. Man Cybern. Syst.* 48, 637–648 (2018). <https://doi.org/10.1109/TSMC.2016.2602543>.
69. Object Management Group: *OMG Systems Modeling Language Version 1.6*. (2019).
70. Booch, G., Rumbaugh, J., Jacobson, I.: *The Unified Modeling Language User Guide*. (1998).
71. MathWorks: Graph with directed edges - MATLAB, <https://www.mathworks.com/help/matlab/ref/digraph.html>, last accessed 2020/11/26.
72. Spivak, D.I.: *Databases are Categories*. (2010).
73. Zeigler, B., Mittal, S., Traore, M.: MBSE with/out Simulation: State of the Art and Way Forward. *Systems*. 6, 40 (2018). <https://doi.org/10.3390/systems6040040>.
74. NAFEMS, INCOSE: What Is Systems Modeling and Simulation? (2019). <https://doi.org/10.1002/9780470403563.ch1>.
75. Mittal, S., Zeigler, B.P., Martin, J.: Implementation of formal standard for interoperability in M&S/systems of systems integration with DEVS/SOA. ... *Command Control* (2009).
76. Abar, S., Theodoropoulos, G.K., Lemarinier, P., O'Hare, G.M.P.: Agent Based Modelling and Simulation tools: A review of the state-of-art software. *Comput. Sci. Rev.* 24, 13–33 (2017). <https://doi.org/10.1016/j.cosrev.2017.03.001>.