PROCEEDINGS A

rspa.royalsocietypublishing.org

Research

Article submitted to journal

Subject Areas:
mathematical modeling, artificial intelligence, category theory

Keywords:
complex systems, system design, automated reasoning, compositionality, applied category theory, operads

Author for correspondence:
John D. Foley
e-mail: foley@metsci.com

Operads for complex system design specification, analysis and synthesis

John D. Foley[1], Spencer Breiner[2], Eswaran Subrahmanian[2,3] and John M. Dusel[1]

[1]Metron, Inc., 1818 Library St., Reston, VA, USA
[2]US National Institute of Standards and Technology, Gaithersburg, MD, USA
[3]Carnegie Mellon University, Pittsburgh, PA, USA

As the complexity and heterogeneity of a system grows, the challenge of specifying, documenting and synthesizing correct, machine-readable designs increases dramatically. Separation of the system into manageable parts is needed to support analysis at various levels of granularity so that the system is maintainable and adaptable over its life cycle. In this paper, we argue that operads provide an effective knowledge representation to address these challenges. Formal documentation of a syntactically correct design is built up during design synthesis, guided by semantic reasoning about design effectiveness. Throughout, the ability to decompose the system into parts and reconstitute the whole is maintained. We describe recent progress in effective modeling under this paradigm and directions for future work to systematically address scalability challenges for complex system design.

1. Introduction

We solve complex problems by separating them into manageable parts [2,86]. Human designers do this intuitively, but details can quickly overwhelm intuition. Multiple aspects of a problem may lead to distinct decompositions and complementary models of a system–e.g. competing considerations for cyberphysical systems [63,87]–or simulation of behavior at many levels of fidelity–e.g. in modeling and simulation [88]–leading to a spectrum of models which are challenging to align. We argue that operads, formal tools developed to compose geometric and algebraic objects, are uniquely suited to separate complex systems into manageable parts and maintain alignment across complementary models.

THE ROYAL SOCIETY
PUBLISHING

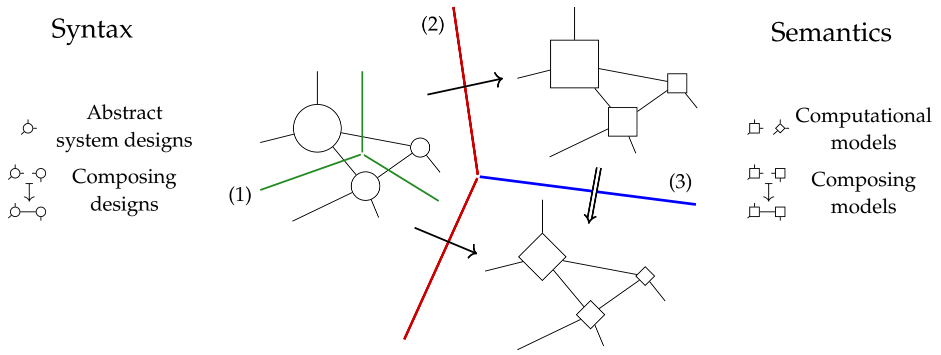


Figure 1: Separating concerns with operads: (1) Composition separates subsystem designs (green boundaries –); (2) Functorial semantics separate abstract systems from computational model instances (red boundaries –); (3) Natural transformations separate and align (blue boundary –) complementary models (□, ◇).

Operads provide three ways to separate concerns for complex systems: (1) designs for subsystems are separated into composable modules; (2) syntactic designs to compose systems are separated from the semantic data that model them; and (3) separate semantic models can be aligned to evaluate systems in different ways. The three relationships are illustrated in Figure 1.

Hierarchical decomposition (Fig. 1, –) is nothing new. Both products and processes are broken down to solve problems from design to daily maintenance. Operads provide a precise language to manage complex modeling details that the intuitive–and highly beneficial–practice of decomposition uncovers, e.g., managing multiple, complementary decompositions and models.

Operads separate the syntax to compose subsystems from the semantic data modeling them (Fig. 1, –). Syntax consists of abstract "operations" to design the parts and architecture of a system. Semantics define how to interpret and evaluate these abstract blueprints. Operad syntax is typically lightweight and declarative. Operations can often be represented both graphically and algebraically (Fig. 4), formalizing intuitive design diagrams. Operad semantics model specific aspects of a system and can range from fast to computationally expensive.

The most powerful way operads separate is by aligning complementary models while maintaining compatibility with system decompositions (Fig. 1, –). Reconciling complementary models is a persistent and pervasive issue across domains [26,27,29,63,66,69,79,84]. Historically, Eilenberg & Mac Lane [34] invented *natural transformations* to align computational models of topological spaces. Operads use natural transformations to align hierarchical decompositions, which is particularly well-suited to system design.

This paper articulates a uniform and systematic foundation for system design and analysis. In essence, the syntax of an operad defines *what can be* put together, which is a prerequisite to decide *what should be* put together. Interfaces define which designs are *syntactically* feasible, but key *semantic* information must be expressed to evaluate candidate designs. Formulating system models within operad theory enforces the intellectual hygiene required to make sure that different concerns stay separated while working together to solve complex design problems.

We note five strengths of this foundation that result from the three ways operads separate a complex problem and key sections of the paper that provide illustrations.

Expressive, unifying meta-language. A meta- or multi-modeling [18] language is needed to express and relate multiple representations. The key feature of operad-based meta-modeling is its focus on coherent mappings between models (Fig. 1, –, –), as opposed to a universal modeling context, like UML, OWL, etc., which is inevitably under or over expressive for particular use cases. Unification allows complementary models to work in concert, as we see in Sec. 5 for function and control. Network operads—originally developed to design systems—were applied to task behavior. This power to unify and express becomes especially important when reasoning across domains with largely independent modeling histories; compare, e.g., [87]. (2(b), 2(d), 3, 4(a), 5)

Minimal data needed for specification. Data needed to set up each representation of a problem is minimal in two ways: (1) any framework must provide similar, generative data; and (2) each level only needs to specify data relevant to that level. Each representation is self-sufficient and can be crafted to efficiently address a limited portion of the full problem. The modeler can pick and choose relevant representations and extend the meta-model as needed. (4, 6(b))

Efficient exploration of formal correct designs. An operad precisely defines how to iteratively construct designs or adapt designs by substituting different subsystems. Constructing syntactically invalid designs is not possible, restricting the relevant design space, and correctness is preserved when moving across models. Semantic reasoning guides synthesis–potentially at several levels of detail. This facilitates lazy evaluation: first syntactic correctness is guaranteed, then multitudes of coarse models are explored before committing to later, more expensive evaluations. The basic moves of iteration, substitution, and moving across levels constitute a rich framework for exploration. We obtain not only an effective design but also formal documentation of the models which justify this choice. (2(b)–(c), 6, 7(e))

Separates representation from exploitation. Operads and algebras provide structure and representation for a problem. Exploitation of this structure and representation is a separate concern. As Herbert Simon noted during his Nobel Prize speech [85]: "…decision makers can satisfice either by finding optimum solutions for a simplified world, or by finding satisfactory solutions for a more realistic world." This is an either-or proposition for a simple representation. By laying the problem across multiple semantic models, useful data structures for each model–e.g. logical, evolutionary or planning frameworks–can be exploited by algorithms that draw on operad-based iteration and substitution. (6, 7(e))

Hierarchical analysis and synthesis. Operads naturally capture options for the hierarchical decomposition of a system, either within a semantic model to break up large scale problems or across models to gradually increase modeling fidelity. (2(a), 5,6(c), 7(a))

(a) Contribution to design literature

There are well-known examples of the successful exploitation of separation. For instance, electronic design automation (EDA) has had decades of success leveraging hierarchical separation of systems and components to achieve very large scale integration (VLSI) of complex electronic systems [16,36,82]. We do not argue that operads are needed for extremely modular domains. Instead, operads may help broaden the base of domains that benefit from separation and provide a means to integrate and unify treatment across domains. On the other hand, for highly integral domains the ability to separate in practically useful ways may be limited [89,102]. The recent applications we present help illustrate where operads may prove useful in the near and longer term; see 7(c) for further discussion.

Compared to related literature, this article is application driven and outward focused. Interest in applying operads and category theory to systems engineering has surged [21,38,53,67,71,94] as part of a broader wave applying category theory to design databases, software, proteins, etc. [22,31,40,46,91–93]. While much of loc. cit. matches applications to existing theoretical tools, the present article describes recent *application driven* advancements and overviews *specific methods* developed to address challenges presented by domain problems. We introduce operads for design to a general scientific audience by explaining what the operads do relative to broadly applied techniques and how specific domain problems are modeled. Research directions are presented with an eye towards opening up interdisciplinary partnerships and continuing application driven investigations to build on recent insights.

(b) Organization of the paper

The present article captures an intermediate stage of technical maturity: operad-based design has shown its practicality by lowering barriers of entry for applied practitioners and demonstrating applied examples across many domains. However, it has not realized its full

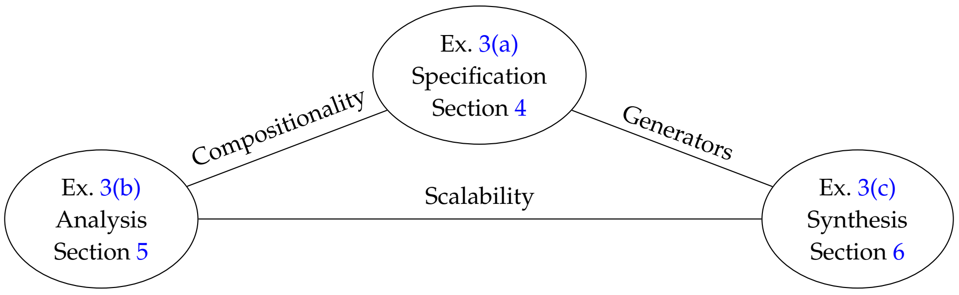


Figure 2: Organization of the paper around applied examples introduced in Sec. 3.

potential as an applied meta-language. Much of this recent progress is not focused solely on the analytic power of operads to separate concerns. Significant progress on explicit specification of domain models and techniques to automatically synthesize designs from basic building blocks has been made. Illustrative use cases and successful applications for design specification, analysis and synthesis organize the exposition.

Section 2 introduces operads for design by analogy to other modeling approaches. Our main examples are introduced in Section 3. Section 4 describes how concrete domains can be specified with minimal combinatorial data, lowering barriers to apply operads. Section 5 concerns analysis of a system with operads. Automated synthesis is discussed in Section 6. Future research directions are outlined in Section 7, which includes a list of open problems.

Notations. Throughout, we maintain the following notional conventions for:

- syntax operads (Fig. 1, left), capitalized calligraphy: $\mathcal{O}$
- types (Fig. 1, edges on left), capitalized teletype: $\mathtt{X}, \mathtt{Y}, \mathtt{Z}, \ldots$
- operations (Fig. 1, nodes on left), uncapitalized teletype: $\mathtt{f}, \mathtt{g}, \mathtt{h}, \ldots$
- semantic contexts (Fig. 1, right), capitalized bold: $\mathbf{Sem}, \mathbf{Set}, \mathbf{Rel}, \ldots$
- functors from syntax to semantics (Fig. 1, arrows across red), capitalized sans serif: $\mathsf{Model}\colon \mathcal{O} \to \mathbf{Sem}$;
- alignment of semantic models via natural transformations (Fig. 1, double arrow across blue), uncapitalized sans serif: $\mathsf{align}\colon \mathsf{Model}_1 \Rightarrow \mathsf{Model}_2$;

2. Applying operads to design

We introduce operads by an analogy, explaining what an operad is and motivating its usefulness for systems modeling and analysis. The theory [64,70,103] pulls together many different intuitions. Here we highlight four analogies or 'views' of an operad: hierarchical representations (tree view), strongly-typed programming languages (API [1] view), algebraic equations (equational view) and system cartography (map-maker's view). Each view motivates operad concepts; see Table 1.

The paradigm of this paper is based on a typed operad, also known as a 'colored operad' [103] or 'symmetric multicategory' [64, 2.2.21]. A typed **operad** $\mathcal{O}$ has:

- A set T of **types**.
- Sets of **operations** $\mathcal{O}(\mathtt{X}_1, \ldots, \mathtt{X}_n; \mathtt{Y})$ where $\mathtt{X}_i, \mathtt{Y} \in T$ and we write $\mathtt{f}\colon \langle \mathtt{X}_i \rangle \to \mathtt{Y}$ to indicate that $\mathtt{f} \in \mathcal{O}(\mathtt{X}_1, \ldots, \mathtt{X}_n; \mathtt{Y})$.
- A specific way to **compose** any operation $\mathtt{f}\colon \langle \mathtt{Y}_i \rangle \to \mathtt{Z}$ with $\mathtt{g}_i\colon \langle \mathtt{X}_{ij} \rangle \to \mathtt{Y}_i$ whose output types match the inputs of f to obtain a composite $\mathtt{f} \circ (\mathtt{g}_1, \ldots, \mathtt{g}_n) = \mathtt{h}\colon \langle \mathtt{X}_{ij} \rangle \to \mathtt{Z}$.

These data are subject to rules [103, 11.2] governing permutation of arguments and assuring that iterative composition is coherent, analogous to associativity for ordinary categories [68, I].

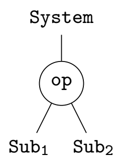
[1] Application Programming Interface

Table 1: The theory of operads draws on many familiar ideas, establishing a dictionary between contexts.

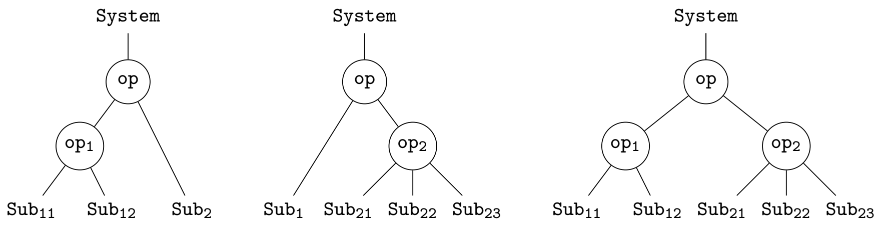
Operads	**Tree**	**API**	**Equational**	**Systems**
Types	Edges	Data types	Variables	Boundaries
Operations	Nodes	Methods	Operators	Architectures
Composites	Trees	Scripts	Evaluation	Nesting
Algebras	Labels	Implementations	Values	Models

(a) The tree view

Hierarchies are everywhere, from scientific and engineered systems to government, business and everyday life; they help to decompose complex problems into more manageable pieces. The fundamental constituent of an operad, called an *operation*, represents a single step in a hierarchical decomposition. We can think of this as a single branching in a labeled tree, e.g.:



Formally, this represents an element $\mathtt{op} \in \mathcal{O}(\mathtt{Sub}_1, \mathtt{Sub}_2; \mathtt{System})$. More generally, we can form new operations—trees—by *composition*. Given further refinements for the two subsystems $\mathtt{Sub}_1$ and $\mathtt{Sub}_2$, by $\mathtt{op}_1$ and $\mathtt{op}_2$, respectively, we have three composites:



(2.1)

Together with the original operation, these represent four views of the same system at different levels of granularity; compare, e.g., [65, Fig. 2]. This reveals an important point: an operad provides a collection of interrelated models that fit together to represent a complex system.

The relationship between models is constrained by the *principle of compositionality*: the whole is determined by its parts *and* their organization. Here, the whole is the root, the parts are the leaves, and each tree is an organizational structure. Formally, *associativity axioms*, which generalize those of ordinary categories, enforce compositionality. For example, composing the left-hand tree above with $\mathtt{op}_2$ must give the same result as composing the center tree with $\mathtt{op}_1$. Both give the tree on the right, since they are built up from the same operations. In day-to-day modeling these axioms are mostly invisible, ensuring that everything "just works", but the formal definitions [103, 11.2] provide explicit requirements and desiderata for modeling languages "under the hood".

Operads encourage principled approaches to emergence by emphasizing the organization of a system. Colloquially speaking, an emergent system is "more than the sum of its parts"; operations provide a means to describe these nonlinearities. This does not explain emergent phenomena, which requires detailed semantic modeling, but begins to break up the problem with separate (but related) representation of components and their interactions. The interplay between these elements can be complex and unexpected, even when the individual elements are quite simple.[2] Compositional models may develop and exhibit emergence as interactions between components are organized, in much the same way as the systems they represent.

[2]For example, diffusion rates (components) and activation/inhibition (interactions) generate zebra's stripes in Turing's model of morphogenesis [99].

(b) The API view

For most applications, trees bear labels: fault trees, decision trees, syntax trees, dependency trees and file directories, to name a few. A tree's labels indicate its semantics either explicitly with numbers and symbols or implicitly through naming and intention.

In an operad, nodes identify operations while edges—called *types*—restrict the space of valid compositions. This is in analogy to type checking in strongly-typed programming languages, where we can only compose operations when types match. In the API view, the operations are abstract method declarations:

```
def op(x1 : Sub1, x2 : Sub2) : System,
def op1(y1 : Sub11, y2 : Sub12) : Sub1,
def op2(z1 : Sub21, z2 : Sub22, z3 : Sub23) : Sub2.
```

Composites are essentially scripted methods defined in the API. For example,

```
def treeLeft(y1 : Sub11, y2 : Sub12, x2 : Sub2) : System
    = op(op1(y1, y2), x2),
```

is a script for left-most tree above. However, the compiler will complain with an invalid syntax error for any script where the types don't match, say

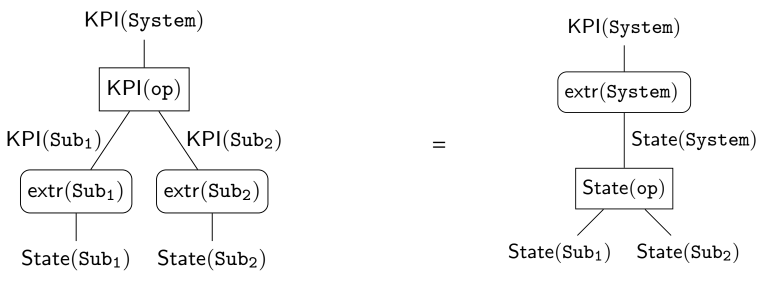
```
def badTree(y1 : Sub11, y2 : Sub12, x2 : Sub2) : System
    = op(x2 ,op1(y1, y2)),
```

If an operad is an API—a collection of abstract types and methods—then an *operad algebra* A is a concrete implementation. An algebra declares: 1) a set of *instances* for each type; 2) a *function* for each operation, taking instances as arguments and returning a single instance for the composite system. That is, $\mathsf{A}\colon \mathcal{O} \to \mathbf{Set}$ has:

- for each type $\mathtt{X} \in T$, a set $\mathsf{A}(\mathtt{X})$ of **instances** of type $\mathtt{X}$, and
- for each operation $\mathtt{f}\colon \langle \mathtt{X_i} \rangle \to \mathtt{Y}$, the function $\mathsf{A}(\mathtt{f})$ **acts** on input elements $a_i \in \mathsf{A}(\mathtt{X_i})$ to obtain a single output element $\mathsf{A}(\mathtt{f})(a_1, \dots, a_n) \in \mathsf{A}(\mathtt{Y})$.

Required coherence rules [103, 13.2] are analogous to the definition of a functor into **Set** [68, I.3]. For example, we might declare a state space for each subsystem, and a function to calculate the overall system state given subsystem states. Alternatively, we might assign key performance indicators (KPIs) for each level in a system and explicit formulae to aggregate them. The main thing to remember is: just as an abstract method has many implementations, an operad has many algebras. Just like an API, the operad provides a common syntax for a range of specific models, suited for specific purposes.

Unlike a traditional API, an operad provides an explicit framework to express and reason about semantic relationships between *different* implementations. These different implementations are linked by type-indexed mappings between instances called *algebra homomorphisms*. For example, we might like to extract KPIs from system state. The principle of compositionality places strong conditions on this extraction: the KPIs extracted from the overall system state must agree with the KPIs obtained by aggregating subsystem KPIs. That is, in terms of trees and in pseudocode:



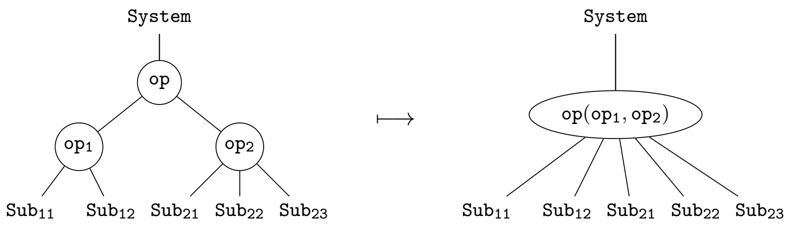
```
KPI(op)(extr(x1), extr(x2))  ==  extr(State(op)(x1, x2)).
```

For any state instances for $\mathtt{Sub}_1$ and $\mathtt{Sub}_2$ at the base of the tree, the two computations must produce the same KPIs for the overall system at the top of the tree. Here $\mathsf{KPI}(\mathtt{op})$ and $\mathsf{State}(\mathtt{op})$ implement $\mathtt{op}$ in the two algebras, while $\mathsf{extr}(-)$ are *components* of the algebra homomorphism to extract KPIs. Similar to associativity, these compositionality conditions guarantee that extracting KPIs "just works" when decomposing a system hierarchically.

(c) The equational view

We have just seen an equation between trees that represent implementations. Because an operad can be studied without reference to an implementation, we can also define equations between abstract trees. This observation leads to another view of an operad: as a system of equations.

The first thing to note is that equations occur within the sets of operations $\mathcal{O}(\mathtt{X}_1, \ldots, \mathtt{X}_n; \mathtt{Y})$; an equation between two operations only makes sense if the input and output types match. Second, if one side of an equation $\mathtt{f} = \mathtt{f}'$ occurs as a subtree in a larger operation $\mathtt{g}$, substitution generates a new equation $\mathtt{g} = \mathtt{g}'$. Two trees are equal if and only if they are connected by a chain of such substitutions (and associativity equations). In general, deciding whether two trees are equal (the word problem) may be intractable. Third, we can often interpret composition of operations as a normal-form computation:



We then compare composed operations directly to decide equality. For example, there is an operad whose operations are matrices. Composition computes a normal form for a composite operation by block diagonals and matrix multiplication,

$$\begin{array}{l} op : n \times (m_1 + m_2) \\ op_1 : m_1 \times (k_{11} + k_{12}) \\ op_2 : m_2 \times (k_{21} + k_{22} + k_{23}) \end{array} \quad \longmapsto \quad \begin{pmatrix} op \end{pmatrix} \cdot \begin{pmatrix} op_1 & 0 \\ 0 & op_2 \end{pmatrix}.$$

Operad axioms constrain composition. For example, the axiom mentioned in 2(a) corresponds to:

$$\begin{pmatrix} op \end{pmatrix} \cdot \begin{pmatrix} op_1 & 0 \\ 0 & I_{m_2} \end{pmatrix} \cdot \begin{pmatrix} I_{k_{11}} & 0 & 0 \\ 0 & I_{k_{12}} & 0 \\ 0 & 0 & op_2 \end{pmatrix} = \begin{pmatrix} op \end{pmatrix} \cdot \begin{pmatrix} I_{m_1} & 0 \\ 0 & op_2 \end{pmatrix} \cdot \begin{pmatrix} op_1 & 0 & 0 & 0 \\ 0 & I_{k_{21}} & 0 & 0 \\ 0 & 0 & I_{k_{22}} & 0 \\ 0 & 0 & 0 & I_{k_{23}} \end{pmatrix}.$$

The key point is that any algebra that implements the operad must satisfy *all* of the equations that it specifies. Type discipline controls which operations can compose; equations between operations control the resulting composites. Declaring equations between operations provides additional contracts for the API. For instance, any unary operation $\mathtt{f} \colon \mathtt{X} \to \mathtt{X}$ (a loop) generates an infinite sequence of composites $\mathsf{id}_\mathtt{X}, \mathtt{f}, \mathtt{f}^2, \mathtt{f}^3, \ldots$. Sometimes this is a feature of the problem at hand, but in other cases we can short-circuit the infinite regress with assumptions like idempotence ($\mathtt{f}^2 = \mathtt{f}$) or cyclicity ($\mathtt{f}^n = \mathsf{id}_\mathtt{X}$) and ensure that algebras contain no infinite loops.

(d) The systems view

When we apply operads to study systems, we often think of an operation $\mathtt{f} \colon \langle \mathtt{X_i} \rangle \to Y$ as a system architecture. Intuitively $\mathtt{Y}$ is the system and the $\mathtt{X}_1, \ldots, \mathtt{X}_n$ are the components, but this is a bit misleading. It is better to think of types as boundaries or interfaces, rather than systems.

Instead, `f` is the system architecture, with component interfaces $\mathtt{X_i}$ and environmental interface `Y`. Composition formalizes the familiar idea [65, Fig. 2] that one engineer's system is the next's component; it acts by nesting subsystem architectures within higher-level architectures.

Once we establish a system architecture, we would like to use this structure to organize our data and analyses of the system. Moreover, according to the principle of compositionality, we should be able to construct a system-level analysis from an operation by analyzing the component-level inputs and synthesizing these descriptions according to the given operations.

The process of extracting computations from operations is called *functorial semantics*, in which a model is represented as a mapping $\mathsf{M}\colon \mathbf{Syntax} \longrightarrow \mathbf{Semantics}$. The syntax defines a system-specific architectural design. Semantics are universal and provide a computational context to interpret specific models. Matrices, probabilities, proofs, and dynamical equations all have their own rules for composition, corresponding to different semantic operads.

The mapping M encodes, for each operation, the data, assumptions and design artifacts (e.g., geometric models) needed to construct the relevant computational representations for the architecture, its components and the environment. From this, the system model as a whole is determined by composition in the semantic context. The algebras (State,KPI) described in 2(b) are typical examples, with syntax $\mathcal{O}$ and taking semantic values in sets and functions. The mappings themselves, called *functors*, map types and operations (`System`, `op`) to their semantic values, while preserving how composition builds up complex operations.

The functorial perspective allows complementary models–e.g. system state vs. KPIs–to be attached to the same design. This includes varying the semantic context as well as modeling details; see Sec. 5 for examples of non-deterministic semantics. Though functorial models may be radically different, they describe the *same system*, as reflected by the overlapping syntax.

In many cases, relevant models are *not* independent, like system state and KPIs. Natural transformations, like the extraction homomorphism in 2(b), provide a means to align ostensibly independent representations. Since models are mappings, we often visualize natural transformations as a two-dimensional cells:

$$\begin{tikzcd}
\mathcal{O} \arrow[r, bend left, "\mathsf{State}", ""{name=U, below}] \arrow[r, bend right, "\mathsf{KPI}"', ""{name=D, above}] & \mathbf{Set}. \arrow[Rightarrow, from=U, to=D]
\end{tikzcd} \tag{2.2}$$

Formal conditions guarantee that when moving from syntax to semantics [103, 13.2] or between representations [64, 2.3.5], reasoning about how systems decompose hierarchically "just works."

Since functors and higher cells assure coherence with hierarchical decomposition, we can use them to build up a desired model in stages, working backwards from simpler models:

$$\begin{tikzcd}
\mathcal{O}_2 \arrow[rrrrr, dashed, "\mathsf{Model}_2\text{ is defined by}"] \arrow[dr, "\mathsf{Extension}_1"'] & & & & & \mathbf{Sem_2} \arrow[dl, "\mathsf{Reduction}_2"] \\
& \mathcal{O}_1 \arrow[rrr, dashed, "\mathsf{Model}_1\text{ is defined by}"] \arrow[dr, "\mathsf{Extension}_2"'] & & & \mathbf{Sem}_1 \arrow[dl, "\mathsf{Reduction}_1"] & \\
& & \mathcal{O}_0 \arrow[r, "\mathsf{Model}_0"'] & \mathbf{Sem}_0 & &
\end{tikzcd}$$

This is a powerful technique for at least two reasons. First, complexity can be built up in stages by layering on details. Second, complex models built at later stages are partially validated through their coherence with simpler ones. The latter point is the foundation for lazy evaluation: many coarse models can be explored before ever constructing expensive models.

Separating out the different roles within a model encourages efficiency and reuse. An architecture (operation) developed for one analysis can be repurposed with strong coherence between models (algebra instances) indexed by the same conceptual types. The syntax/semantics distinction also helps address some thornier meta-modeling issues. For example, syntactic types can distinguish conceptually distinct entities while still mapping to the same semantic entities.

We obtain the flexibility of structural or duck typing in the semantics without sacrificing the type safety provided by the syntax.

3. Main examples

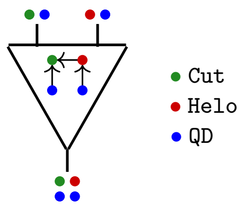
Though operads are general tools [64,70,103], we focus on two classes of operads applied to system design: wiring diagram operads and network operads. These are complementary. Wiring diagrams provide a top-down view of the system, whereas network operads are bottom-up. This section introduces 3 examples that help ground the exposition as in Fig. 2.

(a) Specification

Network operads describe atomic types of systems and ways to link them together with operations. These features enable: (1) specification of atomic building blocks for a domain problem; and (2) bottom up synthesis of designs from atomic systems and links. A general theory of network operads [6,7,73,74] was recently developed under the Defense Advanced Research Projects Agency (DARPA) Complex Adaptive System Composition and Design Environment (CASCADE) program. Minimal data can be used to specify a functor–called a network model [6, 4.2]–which constructs a network operad [6, 7.2] customized to a domain problem.

	Boat	Helo	UAV	QD
Cut	1	1	1	1
Boat			1	1
FW				1
FSAR				1
Helo				1

(a) Examples of carrying relationships in $\mathcal{O}_{Sail}$



(b) Operation $\mathtt{f} \in \mathcal{O}_{Sail}$ to specify carrying

Figure 3: Which types are allowed to carry other types–indicated with 1–specify an operad $\mathcal{O}_{Sail}$; `f` specifies that a `Helo` (●) and a `QD` (●) are carried by a `Cut` (●) and another `QD` (●) is carried on the `Helo` (●).

The first example illustrates designs of search and rescue (SAR) architectures. The domain problem was inspired by the 1979 Fastnet Race and the 1998 Sydney to Hobart Yacht Race and we refer to it as the sailboat problem. It illustrates how network operads facilitate the specification of a model with combinatorial data called a network template. For example, Fig. 3 shows the carrying relationships between different system types to model (e.g., a `Boat` can carry a `UAV` (Unmanned Aerial Vehicle) but a `Helo` cannot). This data specifies a network operad $\mathcal{O}_{Sail}$ whose: (1) objects are lists of atomic system types; (2) operations describe systems carrying other systems; and (3) composition combines carrying instructions. We discuss this example in greater detail in Sec. 4.

(b) Analysis

A wiring diagram operad describes the interface each system exposes, making it clear what can be put together [90,94,104]. The designer has to specify precisely how information and physical quantities are shared among components, while respecting their interfaces. The operad facilitates top-down analysis of a design by capturing different ways to decompose a composite system.

The second example analyzes a precision-measurement system called the Length Scale Interferometer (LSI) with wiring diagrams. It helps illustrate the qualitative features of operads over and above other modeling approaches and the potential to exploit their analytic power to separate concerns. Figure 4 illustrates joint analysis of the LSI to address different aspects of the design problem: functional roles of subsystems and control of the composite system. This analysis example supports these illustrations in Sec. 5.

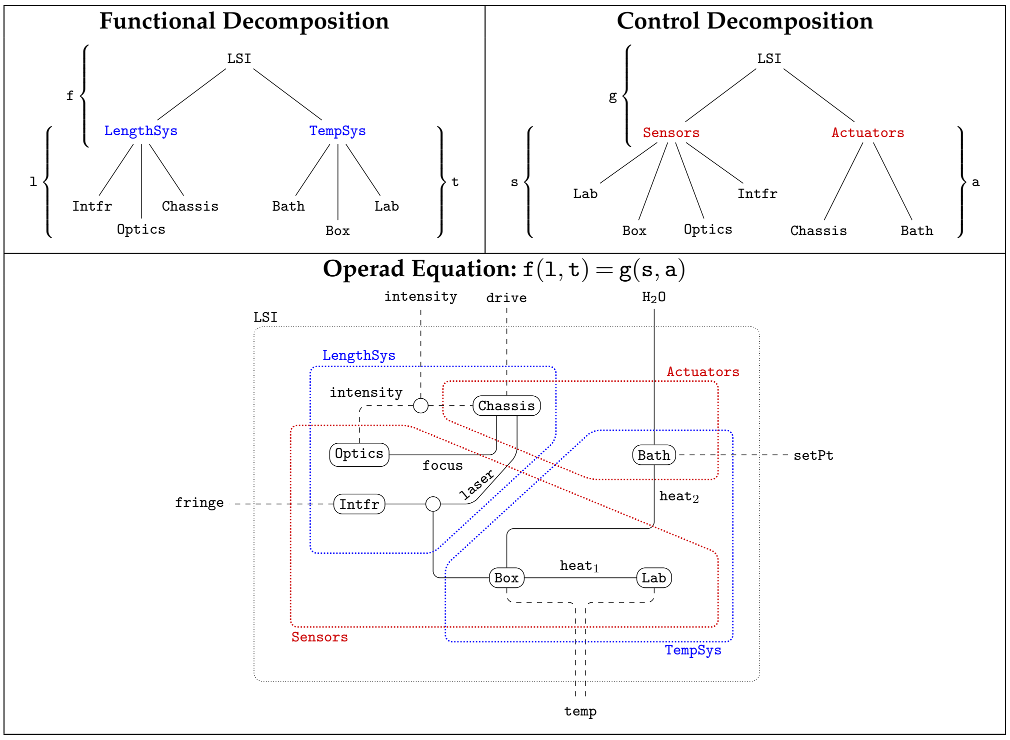
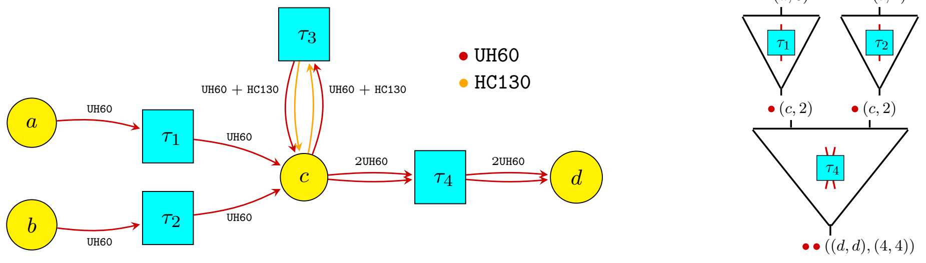


Figure 4: An equation in a wiring diagram operad operations expresses a common refinement of hierarchies.



(a) Specification of primitive tasks := transitions **(b)** Coordinate tasks to compose

Figure 5: Primitive operations are composed for two UH60s (●) to rendezvous at c and maneuver together to d. Each primitive operation is indexed by a transition; types and space-time points must match to compose.

(c) Synthesis

The third example describes the automated design of mission task plans for SAR using network operads. The SAR tasking example illustrates the expressive power of applying existing operads and their potential to streamline and automate design synthesis. Fig. 5a is analogous to Fig. 3, but whereas a sparse matrix specify an architecture problem, here a Petri net is used to model coordinated groups of agents.

For the SAR tasking problem, much of the complexity results from agents' need to coordinate in space and time–e.g. when a helicopter is refueled in the air, as in τ_3 of Fig. 5a. To facilitate coordination, the types of the network operad are systematically extended via a network model whose target categories add space and time dimensions; compare, e.g., [7]. In this way, task plans are constrained at the level of syntax to enforce these key coordination constraints; see, e.g., Fig. 5b where two UH60s at the same space-time point (● $(c, 2)$) maneuver together to d. We describe automated synthesis for this example in Sec. 6.

4. Cookbook modeling of domain problems

In this section we describe some techniques for constructing operads and their algebras, using an example-driven, cookbook-style approach. We emphasize recent developments for network operads and dive deeper into the SAR architecture problem.

(a) Network models

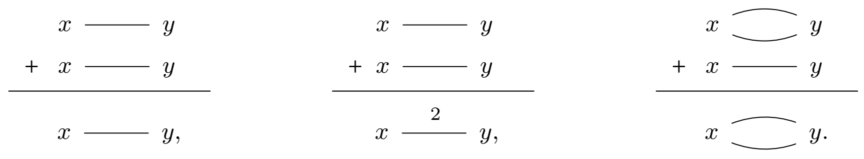
The theory of network models provides a general method to construct an operad $\mathcal{O}$ by mixing combinatorial and compositional structures. Note that this lives one level of abstraction *above* operads; we are interested in *constructing* a language to model systems–e.g. for a specific domain. This provides a powerful alternative to coming up with operads one-by-one. A general construction allows the applied practitioner to cook-up a domain-specific syntax to compose systems by specifying some combinatorial ingredients.

The first step is to specify what the networks to be composed by $\mathcal{O}$ look like. Often this is some sort of graph, but what kind? Are nodes typed (e.g., colored)? Are edges symmetric or directed? Are loops or parallel edges allowed? What about n-way relationships for $n>2$ (hyper-edges)? We can mix, match and combine such combinatorial data to define different *network models*, which specify the system types and kinds of relationships between them relevant to some domain problem. The network model describes the operations we need to compose the networks specific to the problem at hand.

Three compositional structures which describe the algebra of operations. The *disjoint* or *parallel* structure combines two operations for networks with m and n nodes, respectively, into a single operation for networks with $m+n$ nodes. More restrictively, the *overlay* or *in series* structure superimposes two operations to design networks on n nodes. The former structure combines separate operations to support modular development of designs; the latter supports an incremental design process, either on top of existing designs or from scratch. The last ingredient permutes nodes in a network, which assures coherence between different ordering of the nodes. This last structure is often straightforward to specify. If it is not, one should consider if symmetry is being respected in a natural way.

We can distill the main idea behind overlay by asking, what happens when we add an edge to a network? It depends on the kind of network being composed by $\mathcal{O}$:

In a simple graph **but in a labeled graph** **and in a multigraph**



These differences are controlled by a *monoid*[3], which provides each $+$ shown. Above, the monoids are bitwise OR, addition, and maximum, respectively. As a further example, if edge addition is controlled by $\mathbb{Z}/2\mathbb{Z}$ then $+$ will have a toggling effect.

Consider simple graphs. Given a set of nodes $\mathtt{n}$, write $U_\mathtt{n}$ for the set of all undirected pairs $i \neq j$ (a.k.a. simple edges i–j), so that $|U_\mathtt{n}| = \binom{|\mathtt{n}|}{2}$. Then we can represent a simple graph over $\mathtt{n}$ as a $U_\mathtt{n}$-indexed vector of bits $\langle b_{i-j}\rangle$ describing which edges to 'turn on' for a design. Each bit defines whether or not to add an i–j edge to the network and the overlay compositional structure is given by the monoid $\mathsf{SG}(\mathtt{n}) := \mathbf{Bit}^{U_\mathtt{n}}$, whose $+$ is bitwise OR for the product over simple edges–i.e. adding i–j then adding i–j is the same as adding i–j a single time. The disjoint structure $\sqcup : \mathsf{SG}(\mathtt{m}) \times \mathsf{SG}(\mathtt{n}) \longrightarrow \mathsf{SG}(\mathtt{m}+\mathtt{n})$ forms the disjoint sum of the graphs g and h. Finally, permutations act by permuting the nodes of a simple graph. Together, these compositional structures define a network model $\mathsf{SG}\colon \mathcal{S} \to \mathbf{Mon}$ which determines how

[3]A set with a binary operation, usually written $\cdot$ unless the operation is commutative ($m+n=n+m$). A monoid is always associative, $\ell \cdot (m \cdot n) = (\ell \cdot m) \cdot n$ and has a unit e satisfying $e \cdot m = m = m \cdot e$–e.g. multiplication of $n \times n$ matrices.

operations are composed in the constructed network operad; see, Fig. 6 or [6, 3.2, 7.2] for complete technical details.

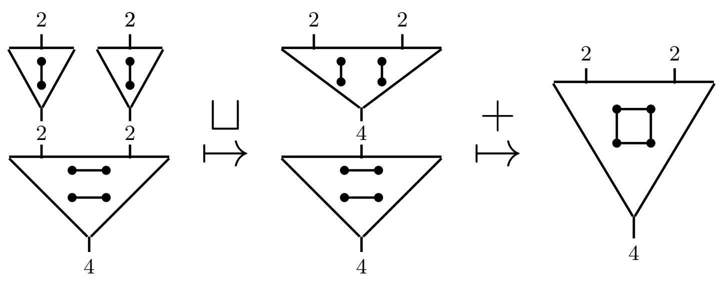


Figure 6: Parallel ($\sqcup$) and in series ($+$) compositional structures define how to combine operations.

This definition has an analogue for $\mathbb{N}$-weighted graphs, $\mathsf{LG}(\mathtt{n}) := (\mathbb{N}, +)^{U_{\mathtt{n}}}$, with overlay given by sum of edge weights and another for multi-graphs, $\mathsf{MG}(\mathtt{n}) := (\mathbb{N}, \max)^{U_{\mathtt{n}}}$, with overlay equivalent to union of multisets; see [6, 3.3, 3.4] for details. More generally, we can label edges with the elements of *any* monoid. Many of these examples are strange—binary addition makes edges cancel when they add—but their formal construction is straightforward; see [6, Thm. 3.1].

Equivalently, we can view the undirected edges in $U_{\mathtt{n}}$ as generators, subject to certain idempotence and commutativity relations: $\mathsf{SG}(\mathtt{n}) := \langle e \in U_{\mathtt{n}} | e \cdot e = e, e \cdot e' = e' \cdot e \rangle$. Here the idempotence relations come from $\mathbf{Bit}$ while the commutativity relations promote the single copies of $\mathbf{Bit}$ for each i–j to a well-defined network model. Similar tricks work for lots of other network templates; we just change the set of generators to allow for new relationships. For example, to allow self-loops, we add loop edge generators $L_{\mathtt{n}} = \mathtt{n} + U_{\mathtt{n}}$ to express relationships from a node i to itself. Likewise, network operads for directed graphs can be constructed by using generators $D_{\mathtt{n}} = \mathtt{n} \times \mathtt{n}$, and one can also introduce higher-arity relationships.

In all cases, the formal definition of a network model assures that all the combinatorial and compositional ingredients work well together; one precise statement of "working well together" is given in [6, 2.3]. Once a *network template*—which expresses minimal data to declare the ingredients for a network model—is codified in a theorem as in [6, 3.1], it can be reused in a wide variety of domains to set up the specifics of composition.

(b) Cooking with operads

The prototype for network operads is a simple network operad, which models only one kind of thing, such as aircraft. The types of a simple network operad are natural numbers, which serve to indicate how many aircraft are in a design. Operations of the simple network operad are simple graphs on some number of vertices. For example, Fig. 6 above shows a simple network operad to describe a design for point-to-point communication between aircraft.

Structural network operads extend this prototype in two directions: (1) a greater diversity of things-to-be-modeled is supported by an expanded collection of types; and (2) more sorts of links or relationships between things are expressed via operations. To illustrate the impact of network templates, suppose we are modeling heterogeneous system types with multiple kinds interactions. For simplicity we consider simple interactions, which can be undirected or directed.

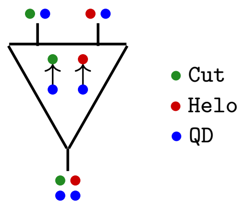
A *network template* need only declare the *primitive* ways system types can interact to define a network model–e.g. a list of tuples (directed : carrying, `Helo`, `Cut`). This data is minimal in two ways: (1) *any* framework must provide data to specify potentially valid interactions; and (2) this approach allows *only* those interactions that make sense upon looking at the types of the systems involved. Thus, interactions must be syntactically correct when constructing system designs.

Presently, we will consider an example from the DARPA CASCADE program: the sailboat problem introduced in 3(a). This SAR application problem was inspired by the 1979 Fastnet Race and the 1998 Sydney to Hobart Yacht Race, in which severe weather conditions resulted in many damaged vessels distributed over a large area. Both events were tragic, with 19 and 6 deaths, respectively, and remain beyond the scale of current search and rescue planning. Various larger assets—e.g. ships, airplanes, helicopters—could be based at ports and ferry smaller search and rescue units—e.g. small boats, quadcopters—to the search area.

Specifically, there were 8 atomic types to model: $P = \{\texttt{Port}, \texttt{Cut}, \texttt{Boat}, \texttt{FW}, \texttt{FSAR}, \texttt{Helo}, \texttt{UAV}, \texttt{QD}\}$. The primary relationship to specify a structural design is various assets carrying another types, so only one kind of interaction is needed: carrying. This relationship is directed; e.g., a cutter (`Cut`) can carry a helicopter (`Helo`) but not the other way around. Specifying allowed relationships amounts to specifying pairs of type $(p, p') \in P \times P$ such that type p' can carry type p; see Fig. 3 for examples. Fig. 3 data is extended to: (1) specify to that `Port` can carry all types other than `Port`, `UAV` and `QD`; (2) conform to an input file format to declare simple directed or undirected interactions, e.g, the JSON format in Fig. 7.

```
{‘colors’ : [‘port’, ‘cut’, ..., ‘qd’],
 ‘directed’ : {
    ‘carrying’: {
        ‘cut’: [‘port’],
        ‘boat’: [‘port’, ‘cut’],
        ...,
        ‘qd’: [‘cut’, ..., ‘helo’] } } }
```

(a) Network template data to specify the operad $\mathcal{O}_{Sail}$



(b) Example operation $\texttt{f} \in \mathcal{O}_{Sail}$

Figure 7: After specifying $\mathcal{O}_{Sail}$, `f` places a `QD` (●) on a `Cut` (●) and another `QD` (●) on a `Helo` (●).

If another type of system or kind of interaction is needed, then the file is appropriately extended. For example, we can include buoys by appending `Buoy` to the array of `colors` and augmenting the relationships in the `carrying` node. Or, we can model the undirected (symmetric) relationship of communication by including an entry such as `‘undirected’: {‘communication’: {‘port‘: [‘cut’, ...], ...}}`. Moreover, modifications to network templates–such as ignoring (undirected : communication) or combining `QD` and `UAV` into a single type–naturally induce mappings between the associated operads [6, 5.8].

(c) Cooking with algebras

Because all designs are generated from primitive operations to add edges, it is sufficient to define how primitive operations act in order to define an algebra. For the sailboat problem, semantics are oriented to enable the delivery of a high capacity for search—known in the literature as search effort [97, 3.1]—in a timely manner. Given key parameters for each asset–e.g. speed, endurance, search efficiency across kinds of target and conditions, parent platform, initial locations–and descriptions of the search environment–e.g. expected search distribution, its approximate evolution over time–the expected number of surviving crew members found by the system can be estimated [97, Ch. 3].

Among these data, the parent platform and initial locations vary within a scenario and the rest describe the semantics of a given scenario. In fact, we assume all platforms must trace their geographical location to one of a small number of base locations, so that the system responds from bases, but is organized to support rapid search. Once bases are selected, the decision problem is a choice of operation: what to bring (type of the composite system) and how to organize it (operation to carry atomic systems). Data for the operational context specifies a particular algebra; see, e.g., Table 2. Just as for the operad, this data is lightweight and configurable.

Related cookbook approaches. Though we emphasized network operads, the generators approach is often studied and lends itself to encoding such combinatorially data with a "template," in a cookbook fashion. The generators approach to "wiring" has been developed

Table 2: Example properties captured in algebra for sailboat problem including time on station (ToS), speed for search (S) and max speed (R), and sweep widths measuring search efficiency for target types person in water (PIW), crew in raft (CIR) and demasted sailboats (DS) adrift.

Type	Cost ($)	ToS (hr)	Speed (kn)		Sweep Width (nmi)		
			S	R	PIW	CIR	DS
`Cut`	200M	∞	11	28	0.5	4.7	8.5
`Boat`	500K	6	22	35	0.4	4.2	7.5
`FW`	60M	9	180	220	0.1	2.2	7.6
`FSAR`	72M	10	180	235	0.5	12.1	16.6
`Helo`	9M	4	90	180	0.5	1.5	4.8
`UAV`	250K	3	30	45	0.5	1.8	4.5
`QD`	15K	4	35	52	0.5	1.5	4.8

into a theory of hypergraph categories [38,41], which induce wiring diagram operads. Explicit presentations for various wiring diagram operads are given in [104]. Augmenting monoidal categories with combinatorially specified data has also been investigated, e.g. in [42].

5. Functorial Systems Analysis

In this section we demonstrate the use of functorial semantics in systems analysis. As in 2(d), a functor establishes a relationship between a syntactic or combinatorial model of a system (components, architecture) and some computational refinement of that description. This provides a means to consider a given system from different perspectives, and also to relate those viewpoints to one another. To drive the discussion, we will focus on the Length Scale Interferometer (LSI) and its wiring diagram model introduced in 3(b).

(a) Wiring diagrams

Operads can be applied to organize both qualitative and quantitative descriptions of hierarchical systems. Because operations can be built up iteratively from simpler ones to specify a complete design, different ways to build up a given design provide distinct avenues for analysis.

Figure 4 shows a wiring diagram representation of a precision measurement instrument called the Length Scale Interferometer (LSI) designed and operated by the US National Institute of Standards and Technology (NIST). Object types are system or component boundaries; Fig. 4 has: 6 components, the exterior, and 4 interior boundaries. Each boundary has an interface specifying its possible interactions, which are implicit in Fig. 4, but define explicit types in the operad.

An operation in this context represents one step in a hierarchical decomposition, as in 2(a). For example, the blue boxes in Fig. 4 represent a functional decomposition of the LSI into length-measurement and temperature-regulation subsystems: $\mathtt{f}\colon \mathtt{LengthSys}, \mathtt{TempSys} \to \mathtt{LSI}$. These are coupled via (the index of refraction of) a `laser` interaction and linked to interactions at the system boundary. The operation `f` specifies the connections between blue and black boundaries.

Composition in a wiring diagram operad is defined by nesting. For this functional decomposition, two further decompositions `l` and `t` describe the components and interactions within `LengthSys` and `TempSys`, respectively. The wiring diagram in Fig. 4 is the composite $\mathtt{f}(\mathtt{l}, \mathtt{t})$.

This approach cleanly handles multiple decompositions. Here the red boxes define a second, control-theoretic decomposition $\mathtt{g}\colon \mathtt{Sensors}, \mathtt{Actuators} \to \mathtt{LSI}$. Unsurprisingly, the system is tightly coupled from this viewpoint, with heat flow to maintain the desired temperature, mechanical action to modify the path of the laser, and a feedback loop to maintain the position of the optical focus based on measured intensity. The fact that these two viewpoints specify the *same* system design is expressed by the equation: $\mathtt{f}(\mathtt{l},\mathtt{t}) = \mathtt{g}(\mathtt{s},\mathtt{a})$; see 2(c) for related discussion.

(b) A probabilistic functor

Wiring diagrams can be applied to document, organize and validate a wide variety of system-specific analytic models. Each model is codified as an algebra, a functor from syntax to semantics (2(d)). For the example of this section, all models have the same source (syntax), indicating that we are considering the same system, but the target semantics vary by application. We have already seen some functorial models: the algebras in 4(c). These can be interpreted as functors from the carrying operad $\mathcal{O}_{Sail}$ to the operad of sets and functions **Set**. Though **Set** is the "default" target for operad algebras, there are many alternative semantic contexts tailored to different types of analysis. Here we target an operad of probabilities **Prob**, providing a simple model of nondeterministic component failure.

The data for the functor is shown in Table 3. Model data is indexed by operations[4] in the domain, an operad $\mathcal{W}$ extracted from the wiring diagram in Fig. 4. The functor assigns each operation to a probability distribution that specifies the chance of a failure in each subsystem, assuming some error within the super-system. For example, the length measurement and temperature regulation subsystems are responsible for 40% and 60% of errors in the LSI, respectively. This defines a Bernoulli distribution $P_{\mathtt{f}}$. Similarly, the decomposition $\mathtt{t}$ of the temperature system defines a categorical distribution with 3 outcomes: `Box`, `Bath` and `Lab`.

Relative probabilities compose by multiplication. This allows us to compute more complex distributions for nested diagrams. For the operation shown in Fig. 4, this indicates that the bath leads to nearly half of all errors ($60\% \times 80\% = 48\%$) in the system.

Operad equations must be preserved in the semantics. Since $\mathtt{f}(\mathtt{l},\mathtt{t}) = \mathtt{g}(\mathtt{s},\mathtt{a})$, failure probabilities of source components don't depend on whether we think of them in terms of functionality or control. For the bath, this relative failure probability is

$$\overbrace{60\%}^{P_{\mathtt{f}}} \times \overbrace{80\%}^{P_{\mathtt{t}}} = 48\% = \overbrace{72\%}^{P_{\mathtt{g}}} \times \overbrace{66.7\%}^{P_{\mathtt{a}}},$$

and five analogous equations hold for the other source components.

Functorial semantics separates concerns: different operad algebras answer different questions. Here we considered *if* a component will fail. The LSI example is developed further in [20, 4] by a second algebra describing *how* a component might fail, with Boolean causal models to propagate failures. The two perspectives are complementary, and loc. cit. explores integrating them with algebra homomorphisms (2(d)).

Table 3: Failure probabilities form an operad algebra for LSI component failure.

$P_{\mathtt{f}}$	$ls \mapsto$	40%		$P_{\mathtt{g}}$	$sn \mapsto$	28%
	$ts \mapsto$	60%			$ac \mapsto$	72%
$P_{\mathtt{l}}$	$in \mapsto$	10%		$P_{\mathtt{s}}$	$lb \mapsto$	21.4%
	$\mathtt{op} \mapsto$	30%			$bt \mapsto$	21.4%
	$ch \mapsto$	60%			$\mathtt{op} \mapsto$	42.9%
$P_{\mathtt{t}}$	$ba \mapsto$	80%			$in \mapsto$	14.3%
	$bx \mapsto$	10%		$P_{\mathtt{a}}$	$ch \mapsto$	33.3%
	$lb \mapsto$	10%			$ba \mapsto$	66.7%

[4]Types and operations, more generally, but the types carry no data in this simple example.

(c) Interacting semantics

Its toy-example simplicity aside, the formulation of a failure model $\mathcal{W} \to \mathbf{Prob}$, as in Table 3 is limited in at least two respects. First, it tells us *which* components fail, but not *how* or *why*. Second, the model is static, but system diagnosis is nearly always a dynamic process. We give a high-level sketch of an extended analysis to illustrate the integration of overlapping functorial models.

The first step is to characterize some additional information about the types in $\mathcal{W}$ (i.e., system boundaries). We start with the dual notions of *requirements* and *failure modes*. For example, in the temperature regulation subsystem of the LSI we have

$$\begin{array}{ccc} T_{\mathtt{laser}} \leq 20.02^\circ\mathrm{C} & \leftrightarrow & T_{\mathtt{laser}} \text{ too high} \\ 19.98^\circ\mathrm{C} \leq T_{\mathtt{laser}} & \leftrightarrow & T_{\mathtt{laser}} \text{ too low} \\ \vdots & & \vdots \end{array}$$

Requirements at different levels of decomposition are linked by traceability relations. These subsystem requirements trace up to the measurement uncertainty for the LSI as a whole. Dually, an out-of-band temperature at the subsystem level can be traced back to a bad measurement in the `Box` enclosure, a short in the `Bath` heater or fluctuations in the `Lab` environment.

Traceability is compositional: requirements decompose and failures bubble up. This defines an operad algebra[5] $\mathsf{Req} : \mathcal{W} \to \mathbf{Rel}^+$. Functoriality expresses the composition of traceability requirement across levels. See [20, 5] discussion of how to link these relations with Table 3 data.

For dynamics, we need *state*. We start with a state space for each interaction among components. For example, consider the `laser` interaction coupling `Chassis`, `Intfr`, and `Box`. The most relevant features of the laser are its vacuum wavelength λ_0 and the ambient temperature, pressure and humidity (needed to correct for refraction). This corresponds to a four-dimensional state-space (or a subset thereof)

$$\mathsf{State}(\mathtt{laser}) \cong \overbrace{[-273.15, \infty)]}^{T_{\mathtt{laser}}} \times \overbrace{[0,\infty)]}^{P_{\mathtt{laser}}} \times \overbrace{[0,1]}^{RH_{\mathtt{laser}}} \times \overbrace{[0,\infty)}^{\lambda_0} \subseteq \mathbb{R}^4.$$

A larger product defines an *external state space* at each system boundary

$$\begin{aligned} \mathsf{State}(\mathtt{TempSys}) = \quad & \mathsf{State}(\mathtt{laser}) \times \mathsf{State}(\mathtt{temp})^2 \times \mathsf{State}(\mathtt{setPt}) \times \mathsf{State}(\mathtt{H_2O}) \\ \mathsf{State}(\mathtt{Box}) = \quad & \mathsf{State}(\mathtt{laser}) \times \mathsf{State}(\mathtt{temp}) \times \mathsf{State}(\mathtt{heat})^2 \\ \vdots \quad & \end{aligned}$$

Similarly, we can define an *internal state space* for each operation by taking the product over all the interactions that appear in that diagram. We can decompose the internal state space in terms of either the system boundary or the components[6]:

$$\begin{aligned} \mathsf{State}(\mathtt{f}) \quad & \cong \quad \mathsf{State}(\mathtt{LSI}) \times \overbrace{\mathsf{State}(\mathtt{laser})}^{\text{hidden variable}} \\ & \cong \quad \mathsf{State}(\mathtt{LengthSys}) \underbrace{\times_{\mathsf{State}(\mathtt{laser})}}_{\text{coupled variable}} \mathsf{State}(\mathtt{TempSys}) \end{aligned}$$

[5]Many operads are defined from ordinary categories using a symmetric monoidal products [56, 7]. If a category carries more than one product, we use a superscript to indicate which is in use. The the disjoint union (+) corresponds to the disjunctive composition "a failure in one component *or* another; soon we will use the Cartesian product $\times$ to consider the conjunctive relationship between "the state of one component *and* the other".
[6]Coupled variables are formalized through a partial product called the pullback, a common generalization of the Cartesian product, subset intersection and inverse image constructions.

$$\begin{tikzcd}
 & \mathbf{Ent} \arrow[d, "\mathsf{Context}"] \\
\mathcal{W} \arrow[ur, dashed, "\mathsf{Val}"] \arrow[r, "\mathsf{State}"'] & \mathbf{Rel}^{\times}
\end{tikzcd}$$

(a) Free Specification

$$\begin{tikzcd}
1 \arrow[r, "\lambda_0\cdot\texttt{fringe}=L_{\texttt{drive}}"] \arrow[d, "\texttt{LSI}"'] & \mathbf{Ent} \arrow[d, "\mathsf{Context}"] \\
\mathcal{W} \arrow[ur, dashed, "\mathsf{Val}"] \arrow[r, "\mathsf{State}"'] & \mathbf{Rel}^{\times}
\end{tikzcd}$$

(b) Top-down requirement

$$\begin{tikzcd}
1 \arrow[r, "\varepsilon_{\texttt{fringe}}\leq\frac{\lambda_0}{8}"] \arrow[d, "\texttt{Intfr}"'] & \mathbf{Ent} \arrow[d, "\mathsf{Context}"] \\
\mathcal{W} \arrow[ur, dashed, "\mathsf{Val}"] \arrow[r, "\mathsf{State}"'] & \mathbf{Rel}^{\times}
\end{tikzcd}$$

(c) Bottom-up requirement

Figure 8: Requirement specification expressed as lifting problems.

The projections from these (partial) products form a relation, and these compose to define a functor $\mathcal{W}\to\mathbf{Rel}^{\times}$:

$$\begin{tikzcd}
\texttt{LSI} & & \texttt{LengthSys},\ \texttt{TempSys} \arrow[ll, "\texttt{f}\in\mathcal{W}"']
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{State}(\texttt{LSI}) & \overbrace{\mathsf{State}(\texttt{f})}^{\in\mathbf{Rel}^{\times}} \arrow[l, "p_0"'] \arrow[r, "\langle p_1{,}p_2\rangle"] & \mathsf{State}(\texttt{LengthSys})\times\mathsf{State}(\texttt{TempSys})
\end{tikzcd}$$

$$\overbrace{\left\langle \dots,\begin{pmatrix}1\\2\\3\end{pmatrix},\begin{pmatrix}8\\9\end{pmatrix},\dots\right\rangle}^{p_0(s)} \longleftarrow\!\!\shortmid \overbrace{\left\langle \dots,\underbrace{\begin{pmatrix}1\\2\\3\end{pmatrix}}_{\texttt{fringe}},\underbrace{\begin{pmatrix}4\\5\\6\\7\end{pmatrix}}_{\texttt{laser}},\underbrace{\begin{pmatrix}8\\9\end{pmatrix}}_{\texttt{H}_2\texttt{O}},\dots\right\rangle}^{s\in\mathsf{State}(\texttt{f})} \longmapsto \left\langle\overbrace{\left\langle \dots,\begin{pmatrix}1\\2\\3\end{pmatrix},\begin{pmatrix}4\\5\\6\\7\end{pmatrix}\right\rangle}^{p_1(s)},\overbrace{\left\langle \begin{pmatrix}4\\5\\6\\7\end{pmatrix},\begin{pmatrix}8\\9\end{pmatrix},\dots\right\rangle}^{p_2(s)}\right\rangle$$

Each requirement $R\in\mathsf{Req}(\texttt{X})$ defines a subset $|R|\subseteq\mathsf{State}(\texttt{X})$, and a state is *valid* if it satisfies all the requirements: $\mathsf{Val}(\texttt{X})=\bigcap_R|R|$. Using pullbacks (inverse image) we can translate validity to internal state spaces in two different ways. External validity (left square) checks that a system satisfies its contracts; joint validity (right square) couples component requirements to define the allowed joint states.

$$\begin{tikzcd}
\mathsf{Val}(\texttt{LSI}) \arrow[d] & \mathsf{XVal}(\texttt{f}) \arrow[l] \arrow[d] & \mathsf{JVal}(\texttt{f}) \arrow[l, dashed] \arrow[r] \arrow[d, tail] & \mathsf{Val}(\texttt{LengthSys})\times\mathsf{Val}(\texttt{TempSys}) \arrow[d] \\
\mathsf{State}(\texttt{LSI}) & \mathsf{State}(\texttt{f}) \arrow[l, "p_0"] \arrow[r, equal] & \mathsf{State}(\texttt{f}) \arrow[r, "\langle p_1{,}p_2\rangle"'] & \mathsf{State}(\texttt{LengthSys})\times\mathsf{State}(\texttt{TempSys})
\end{tikzcd}$$

A requirement model is *sound* if joint validity entails external validity, corresponding to the dashed arrow above. With some work, one can show that these diagrams form the operations in an operad of entailments **Ent**; see [21, 6] for a similar construction. The intuition is quite clear:

$$\begin{array}{rcl}
\text{component reqs.} & \Rightarrow & \text{subsystem reqs.} \\
+\quad \text{subsystem reqs.} & \Rightarrow & \text{system reqs.} \\
\hline
\text{component reqs.} & \Rightarrow & \text{system reqs.}
\end{array}$$

There is a functor $\mathsf{Context}:\mathbf{Ent}\to\mathbf{Rel}^{\times}$, which extracts the relation across the bottom row of each entailment. Noting that the State relations occur in the validity entailment, we can reformulate requirement specification as a *lifting problem* (Fig. 8a): given functors State and Context, find a factorization Val making the triangle commute. The second and third diagrams (Fig. 8b–8c) show how to extend the lifting problem with prior knowledge, in this case a top-level requirement and a known (e.g., off the shelf) component capability.

Finally we are ready to admit dynamics, but it turns out that we have already done most of the work. All that is needed is to modify the spaces attached to our interactions. In particular, we can distinguish between static and dynamic state variables; for the `laser`, T, P and RH are dynamic while λ_0 is static. Now we replace the static values $T, P, RH \in \mathbb{R}$ by functions $T(t), P(t), RH(t) \in \mathbb{R}^T$, thought of as *trajectories* through the state space over a timeline $t \in \tau$. For example, we have

$$\mathsf{Traj}(\texttt{laser}) \subseteq \overbrace{(\mathbb{R}^{\tau})^3}^{T,P,RH} \times \overbrace{\mathbb{R}}^{\lambda_0}.$$

From this, we construct $\mathsf{Traj} : \mathcal{W} \to \mathbf{Rel}^{\times}$ using exactly the same recipe as above. Trajectories and states are related by a pair of algebra homomorphisms inst and const. The first picks out a instantaneous state for each point in time, while the second identifies constant functions, which describe fixed-points of the dynamics:

Global view	Local view

$$\begin{tikzcd}
 & \mathbf{Rel}^{\times} \arrow[dr, "- \times \tau"] \arrow[d, phantom, "\Downarrow \mathsf{inst}"] & \\
\mathcal{W} \arrow[ur, "\mathsf{Traj}"] \arrow[rr, "\mathsf{State}" description] \arrow[rr, bend right, "\mathsf{Traj}"'] & {} \arrow[d, phantom, "\Downarrow \mathsf{const}"] & \mathbf{Rel}^{\times} \\
 & {} &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathsf{Traj}(\texttt{LSI}) \times T \arrow[d, "\mathsf{inst}_{\texttt{LSI}}"] \\
\mathsf{State}(\texttt{LSI}) \arrow[d, "\mathsf{const}_{\texttt{LSI}}"] \\
\mathsf{Traj}(\texttt{LSI})
\end{tikzcd}$$

The problem is that the state space explodes; function spaces are very large. Nonetheless, all of the system integration logic is identical, and using the entailment operad **Ent**, we can build in additional restrictions to limit the search space. In particular, we can restrict attention to the subset of functions that satisfies a particular differential equation or state-transition relationship. This drastically limits the set of valid trajectories, though the resulting set may be difficult to characterize and the methods for exploring it will vary by context.

Related analytic applications. Wiring diagrams have an established applied literature for system design problems, see, e.g., [13,40,90,91,94,100]. More broadly, the analytic strength of category theory to express compositionality and functional semantics is explored in numerous recent applied works–e.g. engineering diagrams [3,4,9,10,17,40,47], Markov processes [5,78], database integration [22,40,76,91,93,95,101], behavioral logic [40,57,83,108], natural language processing [28,51,54], machine learning [39,43], cybersecurity [11–13], quantum computation [35,59,60] and open games [32,48,49].

6. Automated synthesis with network operads

An operad acting on an algebra provides a starting point to automatically generate and evaluate candidate designs. Formally correct designs (operations in some operad) combine basic systems (elements of some algebra of that operad) into a composite system.

(a) Sailboat example

Consider the sailboat problem introduced in 3(a) and revisited in 4(b)–(c). Network operads describe assets and ports carrying each other while algebra-based semantics guided the search for effective designs by capturing the impact of available search effort. To apply this model to automate design synthesis, algorithms explored designs within budget constraints based on costs in Table 2. Exploration iteratively composed up to budget constraints and operational limits on carrying[7]. With these analytic models, greater sophistication was not needed; other combinatorial search algorithms–e.g. simulated annealing–are readily applied to large search spaces. The most effective designs could ferry a large number of low cost search and rescue units–e.g. quadcopters (`QD`)–quickly to the scene–e.g. via helicopters (`Helo`).

[7]Though not used for this application, it turns of that degree limits–e.g. how many quadcopters a helicopter can carry–can be encoded directly into operad operations; the relevant mathematics was worked out in [73].

(b) Tasking example

Surprisingly, network operads—originally developed to design systems—can also be applied to “task" them: in other words, declare their behavior. An elegant example of this approach is given in [7] where “catalyst" agents enable behavioral options for a system.

The SAR tasking problem. The sailboat problem is limited by search: once sailboat crew members are found, their recovery is relatively straightforward. In hostile environments, recovery of isolated personnel (IPs) can become very complex. The challenge is balancing the time criticality of recovery with the risk to the rescuers by judiciously orchestrating recovery teams[8]. Consider the potential challenges of a large scale earthquake during severe drought conditions which precipitates multiple wildfires over a large area. The 2020 Creek Fire near Fresno, CA required multiple mass rescue operations (MROs) to rescue over 100 people in each case by pulling in National Guard, Navy and Marine assets to serve as search and rescue units (SRUs) [52,62]. Though MRO scenarios are actively considered by U.S. SAR organizations, the additional challenge of concurrent MROs distributed over a large area is not typically studied.

In this SAR tasking example, multiple, geographically distributed IP groups compete for limited SRUs. The potential of coordinating multiple agent types—e.g., fire fighting airplanes together with helicopters—to jointly overcome environment risks is considered as well as aerial refueling options for SRUs to extend their range. Depending on available assets, recovery demands and risks, a mission plan may need to work around some key agent types–e.g. refueling assets–and maximize the impact of others–e.g. moving protective assets between recovery teams.

Under CASCADE, tasking operations were built up from primitive tasks that coordinate multiple agent types to form a composite task plan. Novel concepts to coordinate teams of SRUs are readily modeled with full representation of the diversity of potential mission plan solutions.

Network models for tasking. A network model for tasking defines atomic agent types C and possible task plans for each list of agent types. Whereas a network model to design structure $\Gamma\colon \mathcal{S}(C) \to \mathbf{Mon}$ has values that are possible graphical designs, a network model to task behavior $\Lambda\colon \mathcal{S}(C) \to \mathbf{Cat}$ has values that are categories whose morphisms index possible task plans for the assembled types; compare, e.g., [7, Thm. 9]. Each morphism declares a sequence of tasks for each agent–many of which will be coordinated with other agents.

If the system is comprised of only a single UH-60 helicopter, its possible tasks are captured in $\Lambda(\mathtt{UH60})$. In this application, these tasks are paths in a graph describing ‘safe maneuvers.’ For unsafe maneuvers, UH-60s travel in pairs–or perhaps with escorts such as a HC-130 or CH-47 equipped with a Modular Airborne Fire Fighting System (MAFFS). Anything one UH-60 can do, so can two, but not vice versa. Thus there is a proper inclusion $\Lambda(\mathtt{UH60}) \times \Lambda(\mathtt{UH60}) \subsetneq \Lambda(\mathtt{UH60} \otimes \mathtt{UH60})$. Similarly, $\Lambda(\mathtt{UH60}) \times \Lambda(\mathtt{HC130}) \subsetneq \Lambda(\mathtt{UH60} \otimes \mathtt{HC130})$ since once both a UH-60 and HC-130 are present, a joint behavior of midair refueling of the UH-60 by the HC-130 becomes possible. Formally, these inclusions are lax structure maps–e.g. $\Phi_{(\mathtt{UH60},\mathtt{UH60})}\colon \Lambda(\mathtt{UH60}) \times \Lambda(\mathtt{UH60}) \to \Lambda(\mathtt{UH60} \otimes \mathtt{UH60})$, specifies: given tasks for a single UH-60 (left coordinate) and tasks for another UH-60 (right coordinate), define the corresponding joint tasking of the pair. Here the joint tasking is: each UH-60 operates independently within the safe graph. On the other hand, tasks in $\Lambda(\mathtt{UH60} \otimes \mathtt{UH60})$ to maneuver in unsafe regions can not be constructed from independent taskings of each UH-60. Such tasks must be set for some pair or other allowed team–e.g. a CH-47 teamed with an UH-60.

Applying the cookbook: operads. While the above discussion sketches how to specify a network model for tasking, which constructs a network operad [6], these precise details [37] need not concern the applied practitioner[9]. It is sufficient to provide a Petri net as template, from which

[8]The recovery of downed airman Gene Hambleton, call sign Bat 21 Bravo, during the Vietnam War is a historical example of ill-fated SAR risk management. Hambleton’s eventual recovery cost 5 additional aircraft being shot down and 11 deaths; for comparison, a total 71 rescuers and 45 aircraft were lost to save 3,883 lives during Vietnam War SAR [23].

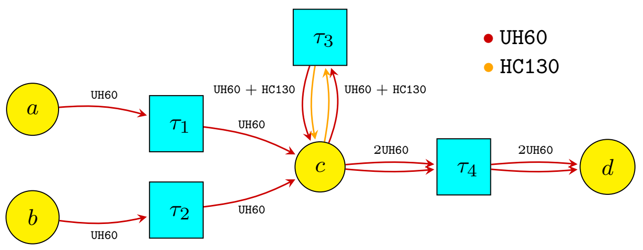
[9]That is, a Petri net specifies the network model $\Lambda\colon \mathcal{S}(C) \to \mathbf{Cat}$ to task behavior. The construction of Λ [37] is similar to the construction described in [7, Thm. 9], but adapted to colored Petri nets whose transitions preserve the number of tokens of each color; see, e.g., Fig. 9a. Compared to [7, Thm. 9], C corresponds to token colors, rather than catalysts [7, Def. 6], and species index discrete coordination locations. Target categories encode allowed paths for each atomic agent type, (cont.)

a network operad is constructed. Whereas a template to design structures defines the basic ways system types can interact, a template to task behavior defines the primitive tasks for agent types C, which are token colors in the Petri net.

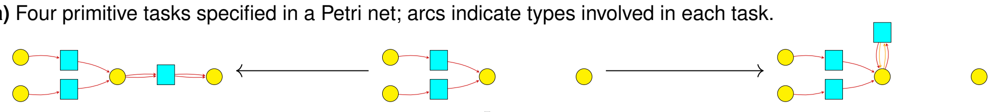
No specification of 'staying put' tasks are needed; these are implicit. All other primitive tasks are (sparsely) declared. For example, each edge of the 'safe graph' for a solo UH-60 declares: (1) a single agent of type `UH60` participates in this 'traverse edge' task; and (2) participation is possible if a `UH60` is available at the source of the edge. Likewise, each edge of the 'unsafe graph' for pairs of UH-60s should declare similar information for pairs, but what about operations to refuel an UH-60 with a HC-130? It turns out that transitions in a Petri net carry sufficient data [7,37] and have a successful history of specifying generators for a monoidal category [8,9,72]. The Petri net Fig. 9a shows examples where, for simplicity, tasks to traverse edges are only shown in the left to right direction. This sparse declaration is readily extended–e.g. to add recovery focused CH-47s, which tested their operational limits to rescue as many as 46 people during the 2020 Creek Fire–C and the set of transitions are augmented to encode the new options for primitive tasks.

This specification of syntax is almost sufficient for the SAR tasking problem and would be for situations where only the sequence of tasks for each agent needs to be planned. When tasking SAR agents, *when* tasks are performed is semantically important because where and how long air-based agents 'stay put' impacts success: (1) fuel burned varies dramatically for ground vs. air locations; (2) risk incurred varies dramatically for safe vs. unsafe locations. For comparison, in a

[9] (cont.), e.g., for Fig. 9a $\Lambda(\texttt{UH60})$ is (freely) generated by objects $\{a, b, c, d\}$ and morphisms $\tau_1 : a \to c$ and $\tau_2 : a \to c$, whereas $\Lambda(\texttt{HC130})$ is just by generated $\{a, b, c, d\}$ since no transition involves a single `HC130`. By describing each target category as an appropriate subcategory of a product of path categories, the symmetric group action is given permuting coordinates, which allows the role of each atomic agent in a task to be specified.



(a) Four primitive tasks specified in a Petri net; arcs indicate types involved in each task.



$$\begin{tikzcd}
{} & {} \arrow[d, Rightarrow, "\text{Network Model}"'] & {} \\
\Lambda(\texttt{UH60} \otimes \texttt{UH60}) & \Lambda(\texttt{UH60}) \arrow[l] \arrow[r] \arrow[d, Rightarrow, "\text{Constraint Matrices}"'] & \Lambda(\texttt{UH60} \otimes \texttt{HC130}) \\
{} & {} & {}
\end{tikzcd}$$

$$M(\texttt{UH60} \otimes \texttt{UH60}) \quad \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & -1 & 1 \end{bmatrix}$$

$$M(\texttt{UH60}) \quad \begin{bmatrix} -1 & 0 & 1 & 0 \\ 0 & -1 & 1 & 0 \end{bmatrix}$$

$$M(\texttt{UH60} \otimes \texttt{HC130}) \quad \begin{bmatrix} -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$M^s(\texttt{UH60} \otimes \texttt{UH60}) \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$M^s(\texttt{UH60}) \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

$$M^s(\texttt{UH60} \otimes \texttt{HC130}) \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

(b) More primitive tasks become possible as available agent types increase. Type update matrices $M(-)$ and target to source constraint matrices $M^s(-)$ translate type changing and matching, resp.

Figure 9: Specified primitive tasks determine an operad $\mathcal{O}_{SAR}$ and a constraint program to explore operations.

ground-based domain without environmental costs, these considerations might be approximately invariant relative to the time tasks occur, and therefore, can be omitted from tasking syntax.

Timing information creates little added burden for building a template–transitions declaring primitive tasks need only be given durations derivable from scenario data–and it is technically straightforward to add a time dimension to the network model.

Constraints from syntax. A direct translation of primitive tasks to decision variables for a constraint program is possible. For syntax, the idea is very simple: enforce type matching constraints on composing operad morphisms. Here we will briefly indicate the original mixed integer linear program developed for SAR tasking; later this formulation was reworked to leverage the scheduling toolkit of the CPLEX optimization software package.

To illustrate the concept, let us first consider the constraint program for an operad to plan tasks without time and then add the time dimension[10]. Operad types are translated to boolean vectors m_j–whose entries capture individual agents at discrete coordination locations. Parallel composition of primitive operations is expressed with boolean vectors Σ_j indexed over primitive tasks for specific agents. Type vectors m_j indicate the coordination location each agent with value one; operation vectors Σ_j indicate which tasks are planned in parallel.

Assuming an operation with task vector Σ_j and source vector m_j, the target is $m_{j+1} = m_j + M\Sigma_j$ where M describes the relationship between source and target for primitive tasks. Rows of M correspond to primitive tasks while columns correspond to individual agents. The target to source constraint for single step of in-series composition is $m_{j+1} \geq M^s \Sigma_{j+1}$ where M^s has rows that give requirements for each primitive task. Here the LHS describes the target and the RHS describes the source. The inequality appears to allow for implicit identities for agents without tasking–e.g. if Σ_j is a zero vector, then $m_{j+1} = m_j$. This constraint prevents an individual agent from being assigned conflicting tasks or 'teleporting' to begin a task.

As seen in Fig. 9b, additional agents: (1) enable more primitive tasks, indexed by Petri net transitions (top two rows); and (2) expand the type vector/matrix column dimension to account for new agent-location pairs and increase the matrix row dimension to account for new tasks (bottom two rows). For example, the first 4 rows of $M(\mathtt{UH60} \otimes \mathtt{UH60})$ correspond to the image of $\Lambda(\mathtt{UH60}) \times \Lambda(\mathtt{UH60})$ in $\Lambda(\mathtt{UH60} \otimes \mathtt{UH60})$. The last row corresponds to new task, τ_4, for the available pair of UH-60s. During implementation, the constraints can be declared task-by-task/row-by-row to sparsely couple the involved agents. Once a limit on the number of steps of in series composition is set–i.e. a bound for the index j is given–a finite constraint program is determined.

Time is readily modeled discretely with tasks given integer durations. This corresponds to a more detailed network model, Λ_t, whose types include a discrete time index; see Fig. 5b for example operations. Under these assumptions, one simply replaces the abstract steps of in series composition with a time index and decomposes M and Σ_j by the duration d of primitive tasks:

$$m_t + \sum_{d=1}^{d_{\max}} M_d \Sigma_{t-d,d} = m_{t+1}; \quad m_{t+1} \geq \sum_{d=1}^{d_{\max}} M_d^s \Sigma_{t+1,d}$$

so that $\Sigma_{t,d}$ describes tasks beginning at time t; the inequality allows for 'waiting' operations. One can also model tasks more coarsely–with $\Lambda_\bullet : \mathbb{N}(C) \to \mathbf{Cat}$–to construct an operad to task counts of agents without individual identity. Then, the type vectors m_j (resp., operation vectors Σ_j) have integer entries to express agent counts (resp., counts of planned tasks) with corresponding reductions in dimensionality. These three levels of network models

$$\begin{tikzcd}
\mathcal{S}(C) \arrow[rr, bend left, "\Lambda_t"] \arrow[rr, bend right, "\Lambda"'] \arrow[d] & \Downarrow & \mathbf{Cat} \\
\mathbb{N}(C) \arrow[urr, bend right, "\Lambda_\bullet"'] & \Downarrow &
\end{tikzcd}$$

[10] Simply increasingly dimensionality is not computationally wise–which was the point of exploring the CPLEX scheduling toolkit to address the time dimension–but this model still serves as a conceptual reference point.

naturally induce morphisms of network operads[11] [6, 6.18] and encode mappings of syntactic variables that preserve feasibility. In particular, the top two levels describe a precise mapping from task scheduling (highest) to task planning (middle). The lowest level $\Lambda_\bullet$ forgets the individual identity of agents, providing a coarser level for planning.

This very simple idea of enforcing type matching constraints is inherently natural[12]. However, further research is needed to determine if this natural hierarchical structure can be exploited by algorithms–e.g. by branching over pre-images of solutions to coarser levels–perhaps for domains were operational constraints coming from algebras are merely a nuisance, as opposed to being a central challenge for SAR planning. For instance, a precise meta-model for planning and scheduling provides a common jumping off point to apply algorithms from those two disciplines.

Applying the cookbook: algebras. Because the operad template defines generating operations, specifying algebras involves: (1) capturing the salient features of each agent type as its internal state; and (2) specifying how these states update under generating morphisms–including, for operads with time, the implicit 'waiting' operations. For the SAR tasking problem, the salient features are fuel level and cumulative probability of survival throughout the mission. Typical primitive operations will not increase these values; fuel is expended or some risk is incurred. The notable exception is refueling operations which return the fuel level of the receiver to maximum. By specifying the non-increasing rate for each agent–location pair, the action of 'waiting' operations are specified. In practice, these data are derivable from environmental data for a scenario so that end users can manipulate them indirectly.

Operational constraints from algebras. Salient features of each agent type are captured as auxiliary variables determined by syntactic decision variables. The values of algebra variables are constrained of update equations–e.g. to update fuel levels for agents with $\max(f_j + F\Sigma_j, f_{\max}) = f_{j+1}$, where $f_{\max}$ specifies max fuel capacities. Having expressed the semantics for generating operations, one can enforce additional operational constraints–e.g. safe fuel levels: $f_{j+1} \geq f_{\min}$.

Extending the domain of application. As noted above, this sparse declaration of a tasking domain is readily extended–e.g. to add a new atomic type or new ways for agents to coordinate. Syntactically, this is amounts to new elements of C or transitions to define primitive tasks. Semantics must capture the impact of primitive operations on state, which can be roughly estimated initially and later refined. This flexibility is especially useful for rapid prototyping of 'what if' options for asset types and behaviors, as the wildfire SAR tasking problem illustrates.

Suppose, for example, that we wanted to model a joint SAR and fire fighting problem. Both domains are naturally expressed with network operads to task behavior. Even if the specification formats were independently developed: (1) each format must encode the essential combinatorial data for each domain; and (2) category theory provides a method to integrate domain data: construct a pushout. Analogous to taking the union of two sets along a common intersection, one identifies the part of the problem common to both domains–e.g. MAFFS-equipped HC-130s and their associated tasks appearing in both domains–to construct a cross-domain model

$$\begin{tikzcd} \mathtt{Spec}_\cap \arrow[r] \arrow[d] & \mathtt{Spec}_{\mathtt{SAR}} \arrow[d] \arrow[dl, phantom, very near start, "\ulcorner"] \\ \mathtt{Spec}_{\mathtt{FF}} \arrow[r] & \mathtt{Spec}_\cup . \end{tikzcd}$$

The arrows in this diagram account for translating the file format for the overlap into each domain-specific format and choosing a specific output format for cross-domain data.

On the other hand, suppose that the machine readable representation of each domain was tightly coupled to algorithms–e.g. mathematical programming for SAR and planning framework for fire fighting. There is no artifact suitable for integrating these domains since expression was prematurely optimized. We describe a general workflow to separate specification from representation and exploitable data structures and algorithms in 7(e).

[11]Strictly speaking, the coarsest (lowest) level is not network model; its domain is a free commutative monoidal category. Nevertheless, a completely analogous construction produces a typed operad fitting into this diagram.
[12]I.e., operad morphisms push forward feasible assignments variables in the domain to feasible assignments in the codomain.

(c) Other examples of automated synthesis

Though network templates facilitate exploration from atoms, how to explore valid designs is a largely distinct concern from defining the space of designs, as discussed in 1.

Novel search strategies via substitution For example, in the DARPA Fundamentals of Design (FUN Design) program, composition of designs employed a genetic algorithm (GA). FUN Design focused on generating novel conceptual designs for mechanical systems–e.g. catapults to launch a projectile. Formulating this problem with network operads followed the cookbook approach: there were atomic types of mechanical components and basic operations to link them.

The operad-based representation provided guarantees of design feasibility and informed how to generalize the GA implementation details. Specifically, composition for atomic algebra elements defined genetic data; crossover produced child data to compose from atoms; and mutation modified parameters of input algebra elements. Crafting a crossover step is typically handled case-by-case while this strategy generalizes to other problems that mix combinatorial and continuously varying data, provided this data is packaged as an operad acting on an algebra. Guarantees of feasibility dramatically reduced the number unfit offspring evaluated by simulation against multiple fitness metrics. Moreover, computational gains from feasibility guarantees increase as the design population becomes more combinatorially complex.

Integrated structure and behavior Large classes of engineering problems compose components to form an 'optimized' network–e.g. in chemical process synthesis, supply chains, and water purification networks [61,75,80,105]. Given a set of inputs, outputs and available operations (process equipment with input and output specification), the goal is to identify the optimal State Equipment Networks (SEN) for behavioral flows of materials and energy. A given production target for outputs is evaluated in terms of multiple objectives such as environmental impact and cost. For example, the chemical industry considers the supply chain, production and distribution network problem [105] systematically as three superstructure optimization problems that can be composed to optimize enterprise level, multi-subsystem structures. Each sub-network structure is further optimized for low cost and other metrics including waste, environmental impact and energy costs. The operadic paradigm would provide a lens to generalize and refine existing techniques to jointly explore structure and behavior.

CASCADE prototyped integrated composition of structure and behavior for distributed logistics applications. Here an explicit resupply plan to task agents was desired. Structural composition was needed to account for the resupply capacity for heterogeneous delivery vehicles and the positioning of distributed resupply depots. Probabilistic models estimated steady state resupply capacities of delivery fleet mixes to serve estimates of demand. First, positioning resupply locations applied hill climbing to minimize the expected disruption of delivery routes when returning to and departing from resupply locations. Second, this disruption estimate was used to adjust the resupply capacity estimate of each delivery asset type. Third, promising designs where evaluated using a heuristic task planning algorithm. At each stage, algorithms focused on finding satisficing solutions which allowed broad and rapid explorations of the design and tasking search space.

Synthesis with applied operads and categories. Research activity to apply operads and monoidal categories to automated design synthesis is increasing. Wiring diagrams have been applied to automate protein design [46,92] and collaborative design [40, Ch. 4] of physical systems employing practical semantic models and algorithms [24,25,106,107]. Software tools are increasingly focused on scaling up computation, e.g. [30,55,58], as opposed to software to augment human calculation, as in [14,60,81], and managing complex domains with commercial-grade tools [22,76,95,101]. Recent work to optimize quantum circuits [35,59] leverages such developments. The use of wiring diagrams to improve computational efficiency via normal forms is explored in [77].

In the next section, we discuss research directions to develop the meta-modeling potential of applied operads to: (1) decompose a problem within a semantic model to divide and conquer; and (2) move between models to fill in details from coarse descriptions. We also discuss how the flow of representations used for SAR–network template, operad model of composition, exploitation data structures and algorithms–could be systematized into a reusable software framework.

7. Toward practical automated analysis and synthesis

In this section, we describe lessons learned from practical experiences with applying operads to automated synthesis. We frame separation of concerns in the language of operads to describe strategies to work around issues raised by this experience. This gives not only a clean formulation of separation but also a principled means to integrate and exploit concerns.

(a) Lessons from automated synthesis

The direct, network template approach facilitates correct and transparent modeling for complex tasking problems. However, computational tractability is limited to small problems–relative to the demands of applications. More research is needed to develop efficient algorithms that break up the search into manageable parts, leveraging the power of operads to separate concerns.

Under CASCADE, we experimented with the CPLEX scheduling toolkit to informally model across levels of abstraction and exploit domain specific information. In particular, generating options to plan, but not schedule, key maneuvers with traditional routing algorithms helped factor the problem effectively. These applied experiments were not systematized into a formal meta-modeling approach, although our prototype results were promising. Specification of these levels–as in Sec. 4–and controlling the navigation of levels using domain-specifics would be ideal.

The FUN DESIGN genetic algorithm approach illustrates the potential operads have to: (1) generalize case-by-case methods[13]; (2) separate concerns, in this case by leveraging the operad syntax for combinatorial crossover and algebra parameters for continuous mutation; and (3) guarantee correctness as complexity grows. Distributed logistics applications in CASCADE show the flexibility afforded by multiple stage exploration for more efficient search.

(b) Formal separation of concerns

We begin by distinguishing *focus* from *filter*, which are two ways operads separate. Focus selects *what* we look at, while filter captures *how* we look at it. These are questions of syntax and semantics, respectively. To be useful, the *what* of our focus must align with the *how* of the filter.

Separation of focus occurs within the syntax operad of system maps. In 2(a), four trees correspond to different views on the same system. We can zoom into one part of the system while leaving other portions black-boxed at a high level. Varying the target type of an operation changes the scope for system composition, such as restricting attention to a subsystem.

Filtering, on the other hand, is semantic; we choose which salient features to model and which to suppress, controlled by the semantic context used to 'implement' the operations. As described in 5(c), the default semantic context is $\mathbf{Set}$ where: (1) each type in the operad is mapped to a set of possible instances for that type; and (2) each operation is mapped to a function to compose instances. Instances or algebra elements for the sailboat problem (Sec. 4) describe the key features of structural system designs. For SAR tasking (Sec. 6), mission plan instances track the key internal states of agents–notably fuel and risk–throughout its execution. Section 5 illustrates alternative semantic contexts as such probability $\mathbf{Prob}$ or relations between sets $\mathbf{Rel}$.

Focus and filter come together to solve particular problems. The analysis of the LSI system in Sec. 5 tightly focuses the syntax operad $\mathcal{W}$ to include only the types and operations from Fig. 4. Formally, this is accomplished by considering the image of the generating types and operations in the operad of port-graphs [20, 3]. This tight focus means semantics need only be defined for LSI components. In each SAR tasking problem of Sec. 6, an initial, source configuration of agent types is given, narrowing the focus of each problem. The SAR focus is much broader because an

[13]In fact, applying genetic algorithms to explore network structures was inspired by the success of NeuroEvolution of Augmenting Topologies (NEAT) [96] to generate novel neural network architectures.

operation to define the mission plan must be constructed. Even so, semantics filter down to just the key features of the problem and how to update them when generating operations act.

Functorial semantics, as realized by an operad algebra $\mathsf{A}\colon \mathcal{O} \to \mathbf{Sem}$, helps factor the overall problem model to facilitate its construction and exploitation. For example, we can construct the probabilistic failure model in Table 3 by normalizing historical failures. First we limit focus from all port-graphs $\mathcal{P}$ to $\mathcal{W}$ then semantics for counts in $\mathbb{N}^{+}$, an operad of counts and sums, are normalized to obtain probabilities in $\mathbf{Prob}$:

$$\begin{tikzcd}
\mathcal{W} \arrow[r] \arrow[d] \arrow[dr, dashed, "\mathsf{A}"] & \mathbb{N}^{+} \arrow[d] \\
\mathcal{P} & \mathbf{Prob}.
\end{tikzcd}$$

The power to focus and filter is amplified because we are not limited by a single choice of how to filter. In addition to *limiting* focus with the source of an operad algebra, we can *simplify* filters. Such natural transformations between functors are 'filters of filters' that align different compositional models precisely–e.g. requirements over state (5(c)) or timed scheduling over two levels of planning (6(b)). In this first case the syntax operad $\mathcal{W}$ stays the same and semantics are linked by an algebra homomorphism (2(d)). In the second case, both the operad and algebra must change to determine simpler semantics–e.g. to neglect the impact of waiting operations, which bound performance. Such precision supports automation to explore design space across semantic models and aligns the ability to focus within each model. By working backward relative to the construction process, we can lift partial solutions to gradually increase model fidelity–e.g. exploring schedules over effective plans. This gives a foundation for lazy evaluation during deep exploration of design space, which we revisit in 7(e).

For a simple but rich example of these concepts working together, consider the functional decomposition $\mathtt{f(l,t)}$ in Fig. 4. We could model the length system $\mathtt{l}$ using rigid-body dynamics, the temperature system $\mathtt{t}$ as a lumped-element model, and super-system $\mathtt{f}$ as a computation (Edlén equation) that corrects the observed `fringe` count based on the measured temperature:

$$\begin{tikzcd}
\{\mathtt{l}\} \arrow[r] \arrow[d, tail] & \mathcal{N}_{\text{mech}} \arrow[r, "\mathsf{Rigid}"] & \mathsf{Dyn} \arrow[d, "\mathsf{Impl}"] \\
\mathcal{W} \arrow[r] & \mathcal{N}_{\text{comp}} \arrow[r, "\mathsf{Edl\acute{e}n}"] \arrow[u, phantom, shift left=8ex, "\Downarrow"] \arrow[d, phantom, shift left=8ex, "\Uparrow"] & \mathsf{Type} \\
\{\mathtt{t}\} \arrow[r] \arrow[u, tail] & \mathcal{N}_{\text{therm}} \arrow[r, "\mathsf{Lump}"] & \mathsf{Dyn} \arrow[u, "\mathsf{Impl}"']
\end{tikzcd} \tag{7.1}$$

The upper and lower paths construct implementations of dynamical models based on the aforementioned formalisms. The center path implements a correction on the data stream coming from the interferometer, based on a stream of temperature data. The two natural transformations indicate extraction of one representation, a stream of state values, from the implementation of the dynamical models. Composition in $\mathcal{W}$ then constructs the two data streams and applies the correction.

A key strength of the operadic paradigm is its genericity: the same principles of model construction, integration and exploitation developed for measurement and SAR apply to all kinds of systems. In principle, we could use the same tools and methodology to assemble transistors into circuits, unit processes into chemical factories and genes into genomes. The syntax and semantics change with the application, but the functorial perspective remains the same. For the rest of this section, we describe research directions to realize such a general purpose vision to decompose a complex design problem into subproblems and support rapid, broad exploration of design space.

(c) Recent advancements, future prospects and limits

Progress driven by applications. Section 4 describes how cookbook-style approaches enable practitioners to put operads to work. Generative data define a domain and compositionality combines it into operads and algebras to separate concerns. Network operads [6,7,73] were developed *in response to the demands of applications* to construct operads from generative data. Section 5 describes rich design analysis by leveraging multiple decompositions of complex systems and working across levels of abstraction. Focusing on a specific applied problem–the LSI at NIST–provided further opportunities for analysis since *model semantics need only be defined for the problem at hand*; see also Eq. 7.1. Progress in streamlining automated synthesis from building blocks is recounted in Sec. 6 where the domain drives coordination requirements to task behavior.

Prospects. If interactions between systems are well-understood (specification) and can be usefully modeled by compositional semantics (analysis), then automated design synthesis leveraging separation for scalability becomes possible. For instance, most references from the end of Sec. 5 correspond to domains that are studied with diagrams that indicate interactions and have associated compositional models. This allows intricate interactions to be modeled–compare, e.g. classical [82] vs. quantum [1,35,59] computing–while unlocking separation of concerns. Cookbook and focused approaches guide practitioners to seek out the minimal data needed for a domain problem–as in the examples presented–but operads for design *requires* compositional models.

Limitations. We note three issues limiting when operad apply: (1) key interactions among systems and components are *inputs*; (2) not all design problems become tractable via decomposition and hierarchy; and (3) there is no guarantee of compositional semantics to exploit. For instance, though the interactions for the n-body problem are understood (1), this does not lend itself to decomposition (2) or exploitable compositional semantics (3). Whitney [102] notes that integral mechanical system design must address safety issues at high power levels due to challenging, long-range interactions. Some aspects of mechanical system design may yield to operad analysis–e.g., bond graphs [17] or other sufficiently "diagrammatic" models–but others may not. Both examples illustrate how overnumerous or long range interaction can lead to (2). Operads can work at the system rather than component level if system properties can be extracted into compositional models. However, operads do not provide a means to extract such properties or understand problems that are truly inseparable theoretically or practically.

(d) Research directions for applied operads

We now briefly overview research directions toward automated analysis and synthesis.

Operad-based decomposition and adaptation. Decomposition, ways a complex operation can be broken down into simpler operations, is a dual concept to the composition of operations. Any subsystem designed by a simpler operation can be adapted: precisely which operations can be substituted is known, providing a general perspective to craft algorithms. To be practical, the analytic questions of *how* to decompose and *when* to adapt subsystems must be answered.

One research direction applies the lens of operad composition to abstract and generalize existing algorithms that exploit decomposition–e.g. to: (1) generalize superstructure optimization techniques discussed in 6(c); extend (2) extend the crossover and mutation steps for the FUN DESIGN work (a), which are global in the sense that they manipulate full designs, to local steps which adapt parts of a design, perhaps driven by analysis to re-work specific subsystems; and (3) explore routing as a proxy for tasking planning, analyzing foundational algorithms like Ford-Fulkerson [44] and decomposition techniques such as contraction hierarchies [45].An intriguing, but speculative, avenue is to attempt to learn how to decompose a system or select subsystems to adapt in a data-driven way, so that the operad syntax constrains otherwise lightly supervised learning. A theoretical direction is to seriously consider the dual role of decomposition, analogous to Hopf and Frobenius algebra [33], and attempt to gain deeper understanding of the interplay of composition and decomposition, eventually distilling any results into algorithms[14].

[14]For example, Bellman's principle of optimality is *decompositional*–i.e. parts of an optimal solution are optimal.

Multiple levels of modeling. The LSI example shows how a system model can be analysed to address different considerations. This sets the stage to adapt a design–e.g. bolster functional risk points and improve control in back and forth fashion–until both considerations are acceptable. Applied demonstrations for SAR tasking suggest a multi-level framework: (1) encoding operational concepts; (2) planning options for key maneuvers; and (3) multistage planning and scheduling to support these maneuvers.

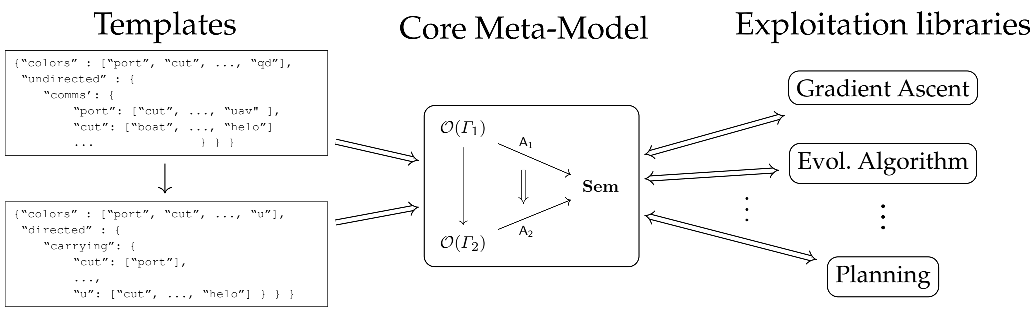
Unifying top-down and bottom-up points of view. We have laid out the analytic–exemplified by wiring diagrams–and synthetic–exemplified by network operads–points of view for complex systems. Even if the goal is practical automated synthesis, scalability issues promote analytic decomposition and abstraction to efficiently reason toward satisficing solutions. Two approaches to unification include: (1) create a combined syntax for analysis and synthesis, a 'super operad' combining both features; (2) act by an analytic operad on the synthetic syntax, extending composition of operations. While the former approach is arguably more unified, the later more clearly separates analysis and synthesis and may provide a constructive approach to former.

(e) Functorial programming with operads

At this point, experience implementing operads for design suggests a software framework. While conceptually simple, this sketch helps clarify the practical role of a precise meta-model.

Rather than working directly with operads to form a core meta-modeling language, cf. [18], a workflow akin to popular frameworks for JavaScript development would put developers in the drivers seat: adopters focus on controlling the flow of data and contribute to an ecosystem of libraries for lower-level data processing. Achieving this requires work before and after the meta-model. First, transferable methods get an applied problem into operads (Fig. 10, left). As in Section 4, this data constructs operads and algebras to form the core meta-model. Core data feeds explicitly exploitable data structures and algorithms to analyze (Sec. 5) and automatically construct (Sec. 6) complex systems (Fig. 10, right). On far the left, end user tools convert intent to domain inputs. Rightmost, libraries access exploitation data structures and algorithms, including those exploiting the syntax and semantics separation or substitution and adaptation. At the center, the core meta-model guarantees that the scruffier ends of the framework exposed to end users and developers are correctly aligned and coherently navigated.

This framework provides significant opportunities to separate concerns compared to other approaches. Foremost, the core model separates syntax from semantics. As noted in 1, applied methods tend to conflate syntax and semantics. For instance, aggregate programming [15] provides: 1) semantics for networked components with spatial and temporal extent; and (2) interactions are proximity-based. The former feature is powerful but limiting: by choosing a single kind of semantics, modeling is wedded to the scales it abstracts well. The individual component scale is not modeled, even syntactically, which would complicate any attempt to align with other models. The latter precludes syntactic declaration of interactions–e.g. to construct architectures not purely based on proximity–and the absolute clarity about what can be put together provided by the operad syntax. Relative to computational efforts to apply operads or monoidal categories,



```
{"colors" : ["port", "cut", ..., "qd"],
 "undirected" : {
    "comms': {
        "port": ["cut", ..., "uav" ],
        "cut": ["boat", ..., "helo"]
        ...                 } } }
```

```
{"colors" : ["port", "cut", ..., "u"],
 "directed" : {
    "carrying": {
        "cut": ["port"],
        ...,
        "u": ["cut", ..., "helo"] } } }
```

Figure 10: A software framework to leverage a meta-model: templates define each level and how to move between, libraries exploit each level, and core meta-model facilitates control across levels.

e.g. [30,55], this sketch places greater emphasis on specification and exploitation: specification of a domain is possible without exposing the meta-model, algorithms searching within each model are treated as black boxes that produce valid designs. Separate specification greatly facilitates set up by experts in the domain, but not the meta-model. Separate exploitation encourages importing existing data structures and algorithms to exploit each model.

(f) Open problems

The software framework just sketched separates out the issues of practical specification, meta-modeling and fast data structures and algorithms. We organize our discussion of open problems around concrete steps to advance these issues. In our problem statements, "multiple" means at least three to assure demonstration of the genericity of the operadic paradigm.

Practical specification. The overarching question is whether the minimal combinatorial data which can specify operads, their algebras and algebra homomorphisms in theory can be practically implemented in software. We propose the following problems to advance the state-of-the-art for network template specification of operads described in Sec. 4:

(i) Demonstrate a specification software package for operad algebras for multiple domains.
(ii) Develop specification software for algebra homomorphisms to demonstrate correctly aligned navigation between multiple models for a single domain.
(iii) Develop and implement composition of specifications to combine multiple parts of a domain problem or integrate multiple domains.

This last point is inline with the discussion of extending a domain in 6(b) and motivates a need to reconcile independently developed specification formats.

(iv) Demonstrate automatic translation across specification formats.

Core meta-model. As a practical matter, state-of-the-art examples exercise general principles of the paradigm but do not leverage general purpose software to encode the meta-model.

(v) Develop and demonstrate reusable middleware to explicitly encode multiple semantic models and maps between them which (a) takes inputs from specification packages; and (b) serves as a platform to navigate models.

We have seen rich examples of focused analysis with wiring diagrams in Sec. 5 and automated composition from building blocks in Sec. 6. Theoretically, there is the question of integrating the top-down and bottom-up perspectives:

(vi) Develop unified foundations to integrate: (a) analytic and synthetic styles of operads; and (b) composition with decomposition.

Potential starting points for these theoretical advancements are described in 7(d). Developing understanding of limitations overviewed in 7(c) requires engagement with a range of applications:

(vii) Investigate limits of operads for design to: (a) identify domains or specific aspects of domains lacking minimal data; (b) demonstrate the failure of compositionality for potentially useful semantics; and (c) characterize complexity barriers due to integrality.

Navigation of effective data structures and algorithms. Lastly, there is the question of whether coherent navigation of models can be made practical. This requires explicit control of data across models and fast data structures and algorithms within specific models. The general-purpose evolutionary algorithms discussed in 6(c) motivate:

(viii) Develop reusable libraries that exploit (a) substitution of operations and instances to adapt designs and (b) separation of semantics from syntax.

SAR tasking experience and prototype explorations for distributed logistics illustrate the need to exploit moving *across* models:

(ix) Develop and demonstrate general purpose strategies to exploit separation across models via hierarchical representation of model fidelity–e.g. example: (a) Structure over behavior; and (b) planning over scheduling.

(x) Quantify the impact of separation of concerns on: (a) computational complexity; and (b) practical computation time.

For this last point, *isolating* the impact of each way to separate concerns is of particular interest to lay groundwork to systematically analyze complex domain problems. Finally, there is question of demonstrating an end-to-end system to exploit the operadic, meta-modeling paradigm.

(xii) Demonstrate systematic, high-level control of iteration, substitution and moving across multiple models to solve a complex domain problem.

(xiii) Develop high-level control framework–similar to JavaScript frameworks for UI–or programming language–similar to probabilistic programming–to systematically control iteration, substitution and movement across multiple models.

8. Conclusion

Operads provide a powerful meta-language to unite complementary system models within a single framework. They express multiple options for decomposition and hierarchy for complex designs, both within and across models. Diverse concerns needed to solve the full design problem are coherently separated by functorial semantics, maintaining compositionality of subsystems. Each semantic model can trade-off precision and accuracy to achieve an elegant abstraction, while algorithms exploit the specifics of each model to analyze and synthesize designs.

The basic moves of iteration, substitution and moving across multiple models form a rich framework to explore design space. The trade-off is that the technical infrastructure needed to fully exploit this paradigm is daunting. Recent progress has lowered barriers to specify domain models and streamline tool chains to automatically synthesize designs from basic building blocks. Key parts of relevant theory and its implementation in software have been prototyped for example applications. Further research is needed to integrate advancements in automatic specification and synthesis with the analytic power of operads to separate concerns. To help focus efforts, we described research directions and proposed some concrete open problems.

Ethics. This article does not present research with ethical considerations.

Data Accessibility. This article has no additional data.

Authors' Contributions. All authors contributed to the development of the review, its text and provided feedback and detailed edits on the work as a whole. JF coordinated the effort and led the writing of Sec. 6. SB led Sec. 5 and co-led Sec. 2, 3 and 4 with JF. ES focused on connections to applications while JD focused on assuring accessible mathematics discussion.

Competing Interests. We declare that we have no competing interests.

Funding. JF and JD were supported by the DARPA Complex Adaptive System Composition and Design Environment (CASCADE) project under Contract No. N66001-16-C-4048.

Acknowledgements. The authors thank John Baez, Tony Falcone, Ben Long, Tom Mifflin, John Paschkewitz, Ram Sriram and Blake Pollard for helpful discussions and two anonymous referees for comments that significantly improved the presentation.

References

1. S. Abramsky and B. Coecke, Categorical quantum mechanics, *Handbook of quantum logic and quantum structures*, 2 (2009), 261–325.
2. C. Alexander, *Notes on the Synthesis of Form*, Harvard University Press, (1964)
3. J. C. Baez, B. Coya and F. Rebro, Props in network theory, *Theor. Appl. Categ.* **33** 25 (2018), 727—783.
4. J. C. Baez and B. Fong, A compositional framework for passive linear networks, *Theor. Appl. Categ.* **33** 38 (2018), 1158—1222.
5. J. C. Baez, B. Fong, and B. S. Pollard, A compositional framework for Markov processes, *J. Math. Phys.* **57** 3 (2016), 033301.
6. J. C. Baez, J. Foley, J. Moeller and B. Pollard, Network models, *Theor. Appl. Categ.* **35** 20 (2020), 700–744.
7. J. C. Baez, J. Foley and J. Moeller, Network models from Petri nets with catalysts, *Compositionality* **1** 4 (2019).
8. J. C. Baez, F. Genovese, J. Master, and M. Shulman, Categories of Nets, *Preprint* (2021). Available as arXiv:2101.04238.
9. J. C. Baez and J. Master, Open Petri nets, *Math. Struct. Comp. Sci.* **30** 3 (2020), 314–341.
10. J. C. Baez, D. Weisbart and A. Yassine, Open systems in classical mechanics, *Preprint* (2021). Available as arXiv:1710.11392.
11. G. Bakirtzis, F. Genovese, and C. H. Fleming, Yoneda Hacking: The Algebra of Attacker Actions, *Preprint*, (2021), available as arXiv:2103:00044.
12. G. Bakirtzis, C. H. Fleming, and C. Vasilakopoulou, Categorical Semantics of Cyber-Physical Systems Theory, *ACM Transactions on Cyber-Physical Systems* (2021, in press), available as arXiv:2010.08003.
13. G. Bakirtzis, C. Vasilakopoulou, and C. H. Fleming, Compositional cyber-physical systems modeling, *Proceedings 3rd Annual International Applied Category Theory Conference 2020 (ACT 2020)*
14. K. Bar, A. Kissinger and J. Vicary, Globular: an online proof assistant for higher-dimensional rewriting, *Log. Methods Comput. Sci.* **14** 1 (2018), 1–16.
15. J. Beal, D. Pianini and M. Viroli, Aggregate programming for the internet of things, *Computer* **48** 9 (2015), 22–30.
16. R. K. Brayton, G. D. Hachtel, C. McMullen and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*, Vol. 2. Springer Science & Business Media, 1984.
17. B. Coya, Circuits, Bond Graphs, and Signal-Flow Diagrams: A Categorical Perspective, PhD thesis, University of California–Riverside, 2018.
18. A. Boronat, A. Knapps, J. Meseguer and M. Wirsing, What is a multi-modeling language? *International Workshop on Algebraic Development Techniques*, Springer, Berlin, Heidelberg (2008), 71–87.
19. S. Breiner, B. Pollard and E. Subrahmanian, Workshop on Applied Category Theory: Bridging Theory and Practice. *Special Publication (NIST SP)* 1249 (2020).
20. S. Breiner, B. Pollard, E. Subrahmanian and O. Marie-Rose, Modeling Hierarchical System with Operads, *Proceedings of the 2019 Applied Category Theory Conference*, (2020) 72–83.
21. S. Breiner, R. D. Sriram and E. Subrahmanian, Compositional models for complex systems, *Artificial Intelligence for the Internet of Everything*, eds. Academic Press, Cambridge Massachusetts (2019) 241–270.
22. K. S. Brown, D. I. Spivak and R. Wisnesky, Categorical data integration for computational science, *Comput. Mater. Sci.* 164 (2019), 127–132.

23. S. Busboom, Bat 21: A Case Study, Carlisle Barracks, PA: U.S. Army War College, (1990).
24. A. Censi, A mathematical theory of co-design, *Preprint* (2015). Available as arXiv:1512.08055.
25. A. Censi, Uncertainty in Monotone Codesign Problems, *IEEE Robotics and Automation Letters* **2** 3 (2017), 1556–1563.
26. M. Chechik, S. Nejati and M. Sabetzadeh, A relationship-based approach to model integration, *Innovations in Systems and Software Engineering* **8** 1 (2012), 3–18.
27. N. Chungoora and R. I. Young, Semantic reconciliation across design and manufacturing knowledge models: A logic-based approach, *Applied Ontology* **6** 4 (2011), 295–295.
28. B. Coecke, M. Sadrzadeh and S. Clark, Mathematical foundations for a compositional distributional model of meaning, *Linguistic Analysis* 36 (2010), 345—384.
29. H. Choi, C. Crump, C. Duriez, A. Elmquist, G. Hager, D. Han, F. Hearl, J. Hodgins, A. Jain, F. Leve, and C. Li, On the use of simulation in robotics: Opportunities, challenges, and suggestions for moving forward, *Proc. Natl. Acad. Sci. U.S.A.* **118** 1 (2021)
30. G. de Felice, A. Toumi and B. Coecke, DisCoPy: Monoidal Categories in Python, *Proceedings 3rd Annual International Applied Category Theory Conference 2020 (ACT 2020).*
31. Z. Diskin and T. Maibaum, Category theory and model-driven engineering: From formal semantics to design patterns and beyond. *Model-Driven Engineering of Information Systems: Principles, Techniques, and Practice* 173 (2014).
32. E. Di Lavore, J. Hedges, and P. Sobociński, Compositional Modelling of Network Games, *Computer Science Logic 2021, Leibniz International Proceedings in Informatics 183* (2021).
33. P. Dusko, Monoidal computer I: Basic computability by string diagrams, *Info. Comp.* **226** (2013), 94–116.
34. S. Eilenberg and S. Mac Lane, General Theory of Natural Equivalences, *Trans. AMS* **58** (1945), 231–294.
35. A. Fagan and R. Duncan, Optimising Clifford Circuits with Quantomatic, *Proceedings 15th International Conference on Quantum Physics and Logic (QPL 2018)*, (2019) 85—105.
36. A. Ferrari and A. Sangiovanni-Vincentelli, System design: Traditional concepts and new paradigms, *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, (1999) 2—12.
37. J. D. Foley, An example of exploring coordinated SoS behavior with an operad and algebra integrated with a constraint program, CASCADE tech report, 2018.
38. B. Fong, The algebra of open and interconnected systems, DPhil thesis, University of Oxford, 2016.
39. B. Fong and M. Johnson, Lenses and learners, *Proceedings of the Eighth International Workshop on Bidirectional Transformations (Bx 2019)*, (2019).
40. B. Fong and D. I. Spivak, *An invitation to applied category theory: seven sketches in compositionality*, Cambridge University Press, 2019.
41. B. Fong and D. I. Spivak, Hypergraph categories, *J. Pure Appl. Algebra* **223** 11 (2019), 4746–4777.
42. B. Fong and D. I. Spivak, Supplying bells and whistles in symmetric monoidal categories, *Preprint* (2020). Available as arXiv:1908.02633.
43. B. Fong, D. I. Spivak and R. Tuyéras, Backprop as functor: A compositional perspective on supervised learning, *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 1–13.
44. L. R. Ford and D. R. Fulkerson, Maximal flow through a network, *Can. J. of Math.* **8** (1956), 399–404.
45. R. Geisberger, P. Sanders, D. Schultes and D. Delling, Contraction hierarchies: Faster and simpler hierarchical routing in road networks, *International Workshop on Experimental and Efficient Algorithms*, Springer, Berlin, Heidelberg (2008), 319–333.
46. T. Giesa, R. Jagadeesan, D. I. Spivak and M. J. Buehler, Matriarch: a python library for materials architecture, *ACS biomaterials science & engineering*, **1** 10 (2015), 1009–1015.
47. D. R. Ghica, A. Jung and A. Lopez, Diagrammatic Semantics for Digital Circuits, *26th EACSL Annual Conference on Computer Science Logic (CSL 2017)*, Leibniz International Proceedings in Informatics, (2017) **82** 24:1–24:16.
48. N. Ghani, C. Kupke, A. Lambert and F. N. Forsberg, Compositional Game Theory with Mixed Strategies: Probabilistic Open Games Using a Distributive Law, *Proceedings of the 2019 Applied Category Theory Conference*, (2020) 95—105.
49. N. Ghani, J. Winschel and P. Zahn, Compositional game theory, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, (2018) 472–481.

50. J. Girard, Linear logic, *Theor. Comput. Sci.* **50** 1 (1987), 1–101.
51. E. Grefenstette, M. Sadrzadeh, S. Clark, B. Coecke, and S. Pulman, Concrete sentence spaces for compositional distributional models of meaning, *Computing meaning*, Springer, Dordrecht (2018) 71-86.
52. J. Guy, 142 Evacuated From Creek Fire by Military Helicopters; Body of Deceased Man Flown to Fresno, *Fresno Bee*, Sept. 8, 2020.
53. K. Gürlebeck, D. Hofmann and D. Legatiuk, Categorical approach to modelling and to coupling of models, *Math. Meth. Appl. Sci.* **40** 3 (2017), 523–534.
54. J. Hedges and M. Sadrzadeh, A generalised quantifier theory of natural language in categorical compositional distributional semantics with bialgebras, *Math. Struct. Comp. Sci.* **29** 6 (2019), 783–809.
55. M. Halter, E. Patterson, A. Baas and J. Fairbanks, Compositional Scientific Computing with Catlab and SemanticModels, *Preprint* (2020). Available as arXiv:2005.04831.
56. C. Hermida. Representable Multicategories, *Advances in Mathematics* **151** 2 (2000), 164-225.
57. P. Johnson-Freyd, J. Aytac, and G. Hulett , Topos Semantics for a Higher-Order Temporal Logic of Actions, *Proceedings of the 2019 Applied Category Theory Conference*, (2020) 161—171.
58. A. Kissinger and J. van de Wetering, PyZX: Large Scale Automated Diagrammatic Reasonin, *Proceedings 16th International Conference on Quantum Physics and Logic (QPL 2019)*, (2020) 229—241.
59. A. Kissinger and J. van de Wetering, Reducing the number of non-Clifford gates in quantum circuits, *Physical Review A*, **102** 2 (2020) 022406.
60. A. Kissinger and V. Zamdzhiev, Reducing the number of non-Clifford gates in quantum circuits, *International Conference on Automated Deduction*, Springer, Cham (2015) 326–336.
61. C. S. Khor, B. Chachuat and N. Shah, A superstructure optimization approach for water network synthesis with membrane separation-based regenerators, *Computers & chemical engineering* **42** (2012), 48–63.
62. R. Kuwada, J. Guy and D. Cooper, Creek Fire roars toward mountain resort towns, after airlift rescues hundreds trapped by flames, *Fresno Bee*, Sept. 6, 2020.
63. E. A. Lee, The past, present and future of cyber-physical systems: A focus on models. *Sensors*, **15** 3 (2015), 4837–4869.
64. T. Leinster, *Higher operads, higher categories*, London Math. Soc. Lec. Note Series, 298 (2003)
65. N. Leveson, The Drawbacks in Using The Term 'System of Systems', *Biomedical Instrumentation & Technology*, (2013), 115–118.
66. C. Lisciandra, and J. Korbmacher, Multiple models, one explanation, *J. Econ. Methodol.* , (2021), 1–21.
67. M. A. Mabrok and M. J. Ryan, Category theory as a formal mathematical foundation for model-based systems engineering, *Appl. Math. Inf. Sci.* **11** (2017), 43–51.
68. S. Mac Lane. *Categories for the Working Mathematician*, 2nd edition, Vol. 5. Springer, (1998)
69. E. Marder and A. L. Taylor, Multiple models to capture the variability in biological neurons and networks, *Nat. Neurosci.* **14** 2 (2011), 133–138.
70. M. Markl, S. Shnider and J. D. Stasheff, *Operads in Algebra, Topology and Physics*, AMS, 2002
71. J. Master, E. Patterson, and A. Canedo, String Diagrams for Assembly Planning, *DIAGRAMS 2020 11th International Conference on the Theory and Application of Diagrams* (2020)
72. J. Meseguer and U. Montanari, Petri nets are monoids, *Inf. Comput.* **88** (1990), 105–155.
73. J. Moeller, Noncommutative network models, *Math. Struct. Comp. Sci.* **30** 1 (2020), 14–32.
74. J. Moeller and C. Vasilakopoulou, Monoidal Grothendieck Construction, *Theor. Appl. Categ.* **35** 31 (2020), 1159–1207.
75. S. M. Neiro and J. M. Pinto, Supply chain optimization of petroleum refinery complexes, *Proceedings of the 4th International Conference on Foundations of Computer-Aided Process Operations*, (2003), 59–72.
76. J. S. Nolan, B. S. Pollard, S. Breiner, D. Anand, and E. Subrahmanian, Compositional models for power systems, *Proceedings of the 2019 Applied Category Theory Conference*, (2020) 72–83.
77. S. M. Patterson, D. I. Spivak and D. Vagner, Wiring diagrams as normal forms for computing in symmetric monoidal categories, *Proceedings of the 2020 Applied Category Theory Conference (ACT 2020)*
78. B. S. Pollard, Open Markov processes: A compositional perspective on non-equilibrium steady states in biologys, *Entropy* **18** (2016), 140.
79. A. Quarteroni, Mathematical models in science and engineering, *Not. Am. Math. Soc.* **56** 1 (2009), 10–19.

80. R. Raman and I. E. Grossmann, Integration of logic and heuristic knowledge in MINLP optimization for process synthesis, *Computers & chemical engineering* **16** 3 (1992), 155–171.
81. D. Reutter and J. Vicary, High-level methods for homotopy construction in associative n-categories, *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, IEEE (2019), 1–13.
82. A. Sangiovanni-Vincentelli, The tides of EDA, *IEEE Design & Test of Computers* **20** 6 (2003), 59–75.
83. P. Schultz and D. I. Spivak, *Temporal Type Theory*, Springer International Publishing, 2019.
84. G. Simon, T. Levendovszky, S. Neema, E. Jackson, T. Bapty, J. Porter and J. Sztipanovits, Foundation for model integration: Semantic backplane, *International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, Vol. 45011, (2012), 1077–1086. American Society of Mechanical Engineers.
85. H. A. Simon, Rational decision making in business organizations, *Am. Econ. Rev.* **69** 4 (1979), 493–513.
86. H. A. Simon, The architecture of complexity, *Facets of systems science*, Springer, (1991), 457–476.
87. M. Sirjani, E. A. Lee, and E. Khamespanah, Verification of Cyberphysical Systems, *Mathematics*, **8** 7, (2020), 1068.
88. J. A. Sokolowski , and C. M. Banks, *Modeling and simulation fundamentals: theoretical underpinnings and practical domains*, John Wiley & Sons, (2010)
89. M. E. Sosa, S. D. Eppinger, and C. M. Rowles, Identifying modular and integrative systems and their impact on design team interactions, *J. Mech. Des.* **125** 2 (2003) 240–252.
90. D. I. Spivak, The operad of wiring diagrams: formalizing a graphical language for databases, recursion, and plug-and-play circuits, *Preprint* (2013). Available as arXiv:1305.0297.
91. D. I. Spivak, *Category Theory for the Sciences*, MIT Press, Cambridge Massachusetts, 2014.
92. D. I. Spivak, T. Giesa, E. Wood and M. J. Buehler, Category theoretic analysis of hierarchical protein materials and social networks, *PloS one* **6** 9 (2011).
93. D. I. Spivak and R. E. Kent, Ologs: a categorical framework for knowledge representation, *PloS one* **7** 1 (2012).
94. D. I. Spivak and J. Tan, Nesting of dynamical systems and mode dependent networks, *J. Complex Networks* **5** 3 (2017), 389–408.
95. D. I. Spivak and R. Wisnesky, Fast Left-Kan Extensions Using The Chase, *Preprint* (2020) Available at www.categoricaldata.net.
96. K. O. Stanley and R. Miikkulainen, Evolving neural networks through augmenting topologies, *Evolutionary Comp.* **10** 2 (2002), 99–127.
97. L. D. Stone, J. O. Royset, and A. L. Wasburn. *Optimal Search for Moving Targets*, Vol. 237. Springer, 2016.
98. M. E. Szabo, *Algebra of proofs*, Studies in logic and the foundations of mathematics, Vol. 88. North-Holland Publishing Company, 1978.
99. A. M. Turing, The Chemical Basis of Morphogenesis, *Philos. Trans. R. Soc. Lond., B, Biol. Sci.* **237** 641 (1952), 37–72.
100. D. Vagner, D. I. Spivak, and E. Lerman, Algebras of open dynamical systems on the operad of wiring diagrams, *Theor. Appl. Categ.* **30** 55 (2015), 1793–1822.
101. R. Wisnesky S. Breiner, A. Jones, D. I. Spivak and E. Subrahmanian, Using category theory to facilitate multiple manufacturing service database integration, *Journal of Computing and Information Science in Engineering* **17** 2 (2017)
102. D. E. Whitney, Physical limits to modularity, (2002)
103. D. Yau, *Colored Operads*, American Mathematical Society, Providence, Rhode Island, 2016.
104. D. Yau, *Operads of Wiring Diagrams*, Vol. 2192. Springer, 2018.
105. H. Yeomans and I. E. Grossmann, A systematic modeling framework of superstructure optimization in process synthesis, *Computers & Chemical Engineering* **23** 6 (1999), 709–731.
106. G. Zardini, N. Lanzetti, M. Salazar, A. Censi, E. Frazzoli, and M. Pavone, Towards a co-design framework for future mobility systems, *Annual Meeting of the Transportation Research Board* (2020).
107. G. Zardini, D. Milojevic, A. Censi, and E. Frazzoli, Co-Design of Embodied Intelligence: A Structured Approach, *Preprint*, (2020), available as arXiv:2011:10756.
108. G. Zardini, D. I. Spivak, A. Censi, and E. Frazzoli, A Compositional Sheaf-Theoretic Framework for Event-Based Systems (Extended Version), *Preprint*, (2020), available as arXiv:2005:04715.