FUNCTORIAL MODEL MANAGEMENT

Breiner, Spencer (1); Pollard, Blake (1,2); Subrahmanian, Eswaran (1,2)

1: National Institute of Standards and Technology (NIST); 2: Carnegie Mellon University

ABSTRACT

In this paper we use formal tools from category theory to develop a foundation for creating and managing models in systems where knowledge is distributed across multiple representations and formats. We define a class of models which incorporate three different representations---computations, logical semantics, and data--as well as model mappings (functors) to establish relationships between them. We prove that our models support model merge operations called colimits and use these to define a methodology for model integration.

Keywords: Semantic data processing, Systems Engineering (SE), Information management, Model management, Data integration

Contact:
Breiner, Spencer
National Institute of Standards and Technology (NIST)
Information Technology Lab
United States of America
spencer.breiner@nist.gov

Cite this article: Breiner, S., Pollard, B., Subrahmanian, E. (2019) ‘Functorial Model Management’, in *Proceedings of the 22nd International Conference on Engineering Design (ICED19),* Delft, The Netherlands, 5-8 August 2019. DOI:10.1017/dsi.2019.202

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

1 INTRODUCTION

In this paper we consider the problem of model management (MM) in systems of distributed knowledge representation. We intend the term “model” to apply liberally, including more-or-less any structured mathematical or computational representation of the world, ranging from databases to differential equations. Such representations are necessarily scoped by particular domains and demands, and when new applications arise they often require the extension and integration of existing models. It is inevitable, then, that practical systems for knowledge representation must provide a space for interaction between different models built for different purposes using different formalisms.

Towards this end we argue for the use of category theory (CT) as a foundation for MM. CT is the study of compositional processes, providing a mathematical theory of the way that complicated systems are built up from simpler pieces. Drawing on applications in programming language theory, formal logic and databases, we define a class of models which incorporate computation, data and semantic modeling into one package, as well as model transformations which relate them to one another. We prove that these structures support a minimal merge operation called a pushout, and use this to develop a methodology for model integration and evolution, illustrated with an example from collaborative design.

Section 2 gives a brief review of model management applications in engineering. Section 3 is short but technical, giving formal definitions for our models and model transformations as well as a proof that these are closed under a family of algebraic integration operations called colimits. To make things more concrete, Section 4 presents an example of such a model, which is used in Section 5 to motivate a general methodology for model integration and evolution.

2 MODEL MANAGEMENT

Model management (MM) is a sub-discipline of knowledge representation concerned with the problem of organizing and synchronizing multiple overlapping representations. In complex contexts like science, engineering and commerce we are inescapably faced with the need to apply many different models built for many different purposes expressed in many different formalisms to solve a given problem. For example, in Chandrasegaran *et al.* (2013) the authors identify 30 different *classes* of representations (e.g., structural analysis) involved in a typical product-design process, and each of these encompasses a variety of more specific model types (e.g., mechanical, elastic, finite-element). Moreover, the information which goes into these representations must often be assembled from many different sources and stored in many different formats.

With such a broad purview, it is not surprising that elements of MM have already been explored in engineering design. One early example is the ASCEND system (Piela *et al.*, 1991), which was developed to model chemical engineering processes by composing unit operations. Using an object-oriented approach, components could be modeled independently and then combined to construct global optimization models. These could then be analyzed for unit consistency and degrees of freedom before passing to an appropriate solver based on the structure of the resulting model.

Around the same period, the *n*-dim group (Levy *et al.*, 1993; Subrahmanian *et al.*, 1997) developed a prototype-based system which used graphs and graph transformations to manage information exchange and internal coherence for several collaborative engineering projects. This work later influenced Wynn *et al.* (2009), who developed a configurable diagrammatic modeling platform called P3 for the creation and modeling of prototype engineering solutions.

There are other areas of engineering design where MM has been recognized as an important issue, although the connection with more abstract model integration may not be recognized. One example is the management of configurable products and product variety (Männistö and Sulonen, 1999), where different configurations with different model schemas must be compared to one another. More recently, Eckert *et al.* (2017) has identified data integration for product and process designs as an obstacle in contemporary engineering design.

A common feature in all these problems is the need to identify overlapping concepts and data which occur in distinct contexts. A rather different approach to addressing such problems has emerged from the field of database management, specifically with regards to problems like data integration,

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

federation and migration. Bernstein *et al.* (2000) sets out a vision for addressing these challenges through the use of model mappings, leading to the creation of a model management group at Microsoft Research (Microsoft Research, 2001–2011) which explored many practical and theoretical aspects of these problem from 2001 to 2011.

Category theory (CT) is the mathematical study of compositional mappings. Once the MM problem has been framed in these terms of mappings, CT becomes a natural candidate with which to formulate a solution. In fact, this was already recognized in some of the group's earliest work (Alagić and Bernstein, 2001). While there is some overlap between our work and theirs (notably, the use of pushouts for model integration), the earlier work assumes a category of models and mappings as given and considers the properties it might exhibit; here we define those structures for ourselves and verify the desirable properties directly.

Our approach is closer to the recent paper Schultz *et al.* (2016), where schemas and mappings are defined as categories and functors. This uses the fact that CT is highly expressive meta-modeling language, able to support translations from database schemas (Rosebrugh and Wood, 1992; Spivak, 2012), OWL ontologies (Patterson, 2017), UML class diagrams (Sarala *et al.*, 2018) Entity-Relation diagrams (Johnson *et al.*, 2002) and more. We can then use mappings between categories (called *functors*) to express relationships between any of these formats.

Our work here builds on Schultz *et al.* (2016) in two main respects. First, as we saw above, practical MM requires connecting to tools and solvers, so we place a much stronger emphasis on the role of computation in our models. Second, we encapsulate computations, semantic models and data into a single package, providing a more coherent interface for managing all three in a uniform way.

3 A CATEGORY OF INTEGRATED MODELS

In this section we provide formal definitions for our models and model tranformations. These incorporate three distinct types of representation: computations, semantic/ontological modeling and data. We also prove that these structures are closed under algebraic integration operators called colimits (specifically, pushouts).

Our approach is based on a branch of mathematics called category theory (CT). By defining our models in terms of such mathematical structures, we can guarantee that operations like model integration and composition of transforms are well-behaved. Unfortunately, the material is quite technical and a proper introduction is beyond the scope of this paper. However, this section may be omitted from a first reading without interrupting the flow of the discussion, and we will use an example in the next section to explain the intuitions underlying our definitions.

Definition 1. *Fix a Cartesian closed category* **Type** *and a functor* **eval** : **Type** $\to$ **Set**. *In this context, a* model $\mathbb{S}$ *is defined by:*

- *a (finitely-presented) category*[1] $\mathbb{S}$,
- *a projection functor*[2] $\pi : \mathbb{S} \to \mathbb{2}$,
- *an implementation functor* **impl** : $\mathbb{S}_1 := \pi^{-1}(1) \to$ **Type**,
- *and a data functor* **data** : $\mathbb{S} \to$ **Set**,
- *making the diagram to the right commute.*

$$\begin{tikzcd}
\mathbb{S}_1 \arrow[r, phantom, "\subseteq"] \arrow[d, "\mathbf{impl}"'] & \mathbb{S} \arrow[d, "\mathbf{data}"] \\
\mathbf{Type} \arrow[r, "\mathbf{eval}"'] & \mathbf{Set}.
\end{tikzcd}$$

Because our models are defined in terms of categorical structures, we can easily define model transformations using functors and natural transformations.

Definition 2. *A* model transformation $F = (F, \varphi) : \mathbb{S} \to \mathbb{T}$ *consists of:*

- *a functor* $F : \mathbb{S} \to \mathbb{T}$ *commuting over* $\mathbb{2}$
- *which agrees on implementation:* $\mathbf{impl}_{\mathbb{S}} = F.\mathbf{impl}_{\mathbb{T}}$
- *and a natural transformation* $\varphi : \mathbf{data}_S \Rightarrow F.\mathbf{data}_T$ *which restricts to the identity on types.*

Since both functors and natural transformations can be composed, it is clear that these mappings form a category which we denote **Mod**.

[1] By abuse of notation, we use the same symbol $\mathbb{S}$ to refer to a model and its underlying schema.

[2] Here $\mathbb{2}$ is the category $\{0 \leq 1\}$ with two objects and one non-identity arrow.

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

The model integration approach we describe in the Section 5 relies on the following theorem, which states that our models are closed under a class of algebraic operations called *pushouts* (more generally *colimits*). These allow us to construct a merged model from two (or more) component models, but only once the overlapping elements of the models (schemas, data and computations) have been specified.
Theorem 1. *The category of models is closed under finite colimits.*

*Proof Sketch.*We use the standard fact (Adámek and Rosicky, 1994) that both schemas (finitely-presented categories) and data instances (**Set**-valued functors) are, independently, closed under colimits. We can then knit these together using a second construction called a (left) Kan extension (denoted Σ).

We give an operational construction of the pushout; it's correctness follows by unwinding the definitions. Other colimits can be constructed from pushouts using standard reasoning (Awodey, 2010).

By assumption, we begin with two transformations $\mathbb{S} \xleftarrow{(L,\lambda)} \mathbb{O} \xrightarrow{(R,\rho)} \mathbb{T}$. First, construct a pushout of schemas $\mathbb{P} = \mathbb{S} \underset{\mathbb{O}}{\oplus} \mathbb{T}$, which comes equipped with two additional functors $\mathbb{S} \xrightarrow[I]{} \mathbb{P} \xleftarrow[J]{} \mathbb{T}$. We can then use the universal property of the pushout to define the functors $\pi : \mathbb{P} \to \mathbb{2}$ and **impl** : $\mathbb{P}_1 \to$ **Type**.

Next, we lift all of our data instances and mappings to the common schema $\mathbb{P}$ using Kan extensions. Formally, this relies on both the functoriality $(*)$ and the counit of the Kan adjunction $\epsilon : \Sigma_L(L.X) \Rightarrow X$. For example, starting from the natural transformation $\lambda : \mathbf{data}_{\mathbb{O}} \Rightarrow L.\mathbf{data}_{\mathbb{S}}$ over $\mathbb{O}$, we construct the following transformation over $\mathbb{P}$ (and similarly for ρ):

$$\Sigma_{L.I}(\mathbf{data}_{\mathbb{O}}) \xRightarrow{\Sigma_{L.I}(\lambda)} \Sigma_{L.I}(L.\mathbf{data}_{\mathbb{S}}) \overset{(*)}{=} \Sigma_I(\Sigma_L(L.\mathbf{data}_{\mathbb{S}})) \xRightarrow{\Sigma_I(\epsilon)} \Sigma_I(\mathbf{data}_{\mathbb{S}}) \tag{1}$$

With these in one place we can construct the pushout of the data instances (i.e., **Set**-valued functors) to form an integrated data set on the joint schema. This comes equipped with maps relating the original data to that in the integrated model. These define the necessary inclusion mappings to complete the pushout in **Mod**.

We note one particularly important feature of this construction: the use of *labeled nulls* (also called Skolem variables). These allow us to define canonical database extensions even in the absence of information which is required by the data model. For example, suppose we extend a schema by adding a new, unrelated attribute to one table. We cannot infer these values from the original data, so instead we create a labeled null to stand in its place. The same problem occurs whenever we merge schemas which contain independent information. In contrast to the standard use of database nulls, with one global value, here each variable has an independent identity which can participate (symbolically) in constraints and calculations.

4 AN EXAMPLE FROM ENGINEERING DESIGN

In this section we explain the categorical structures defined in the previous section by reference to a concrete example from collaborative engineering. In this scenario two teams are collaborating on a design project, with one responsible for the material design of a component and the other for its structure. In this section we focus on the structural design team, with an eye towards integration in Section 5.

A category is a mathematical structure with two classes of elements called *objects* and *arrows*, which are analogous to the nodes and edges in a directed graph. Categories extend the geometric structure of graphs with additional algebraic operations that let us combine objects and arrows in different ways. For our purposes here, the most important will be composition $f.g$ and pairing $\langle f,g\rangle$.

Figure 1 shows a database fragment from the structural engineering team. It contains two tables, `Build` and `Material`, which will correspond to two objects in our categorical schema. Furthermore, the last column of the `Build` table establishes a foreign-key relationship between the two (indicated here by '`@`'), corresponding to an arrow `material` : `Build` → `Material`.

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

While some columns store foreign keys, most store concrete data values. These also correspond to arrows in the schema, but ones which target abstract datatypes rather than other tables. For example, the second column of the Material table defines an arrow $\texttt{name} : \texttt{Material} \to \texttt{String}$.

Consequently, datatypes like String are also objects in our schema. In addition to simple types like String, schemas may include complex types like lists, user-defined classes (as in object-oriented programming) and outside data formats. Here we will assume a binary blob datatype CAD as well as two locally-implemented classes MatProp and TestResults which contain materials-property profiles and component testing results, respectively.

The goal of our hypothetical team is to produce an optimization algorithm which will accept a preliminary CAD design for the part along with a material profile and return a new geometry optimized to the given properties. Computations also appear in our schema, as arrows from datatypes to datatypes. Here the structural team's output corresponds to an arrow $\texttt{optimize} : \texttt{CAD} \times \texttt{MatProp} \to \texttt{CAD}$.

Thus, all in all, our schema contain two kinds of objects–*entities* (tables) and *types*–and three kinds of arrows: *correspondences* (entity-to-entity), *attributes* (entity-to-type) and *computations* (type-to-type). Formally, our definition encodes these distinctions as a typing map $\pi : \mathbb{S} \to \mathbb{2}$, where the target category $\mathbb{2}$ has exactly two objects and three arrows, corresponding to the 2+3 italicized terms listed above.

One advantage of collecting computations and entities in one place is that we can encode data constraints that incorporate both. For example, we would like to formalize the workflow requirement of the structural engineers' design project: the final model for a Build should be determined by applying the optimize algorithm to the prelim model and the build's material.

The problem is that the structural team has no way of linking theMaterial entity associated with a Build to the MatProps datatype required by the optimize algorithm. If the team had access to an additional attribute $\texttt{matProps} : \texttt{Material} \to \texttt{MatProps}$, though, we could formulate the workflow requirement using composition and pairing:

$$\texttt{build.final} = \texttt{build.}\langle\texttt{prelim,material.matProps}\rangle\texttt{.optimize} \tag{2}$$

The final schema is shown in Figure 2, with entities at the top and types at the bottom. The schema is an abstract syntax which identifies the elements of the domain (vocabulary) and how they fit together (grammar). Complex terms built from composition, pairing and other operations form the statements of the language, and these can be analyzed using equations, as above, or through more complex relationships.

In fact, there are two schemas here. $\mathbb{S}$ includes everything except the dashed matProps arrow, and represents the information that the structural engineering team already has. $\mathbb{S}'$ includes both solid and dashed arrows, and represents the information that they want. The two are connected by functor $X : \mathbb{S} \to \mathbb{S}'$ which identifies one schema as a subgraph of the other.

The remaining elements of a model concern its semantics: what is the meaning of the elements that appear in a schema. First of all, CT provides a variety of semantic contexts which correspond to

Material	name	composition	matProps
@mat1	IronChromium	Fe-14Cr-4Ni-4	???
@mat2	AluminumSilicon	AlSi-10Mg	???
@mat3	TitaniumAluminum	Ti-6Al-4V	???

Build	date	batchID	prelim	final	test	material
@build1	1/25/19	3.4.1.20.19	⟨data⟩	⟨data⟩	{wear=fail,...}	@mat1
@build2	1/28/19	3.4.1.20.19	⟨data⟩	⟨data⟩	{wear=pass,...}	@mat1
@build3	2/15/19	5.2.2.2.19	⟨data⟩	⟨data⟩	{wear=fail,...}	@mat2
@build4	3/2/19	3.7.2.15.19	⟨data⟩	⟨data⟩	{wear=pass,...}	@mat1
@build5	3/7/19	5.2.2.2.19	⟨data⟩	⟨data⟩	{wear=pass,...}	@mat2
@build6	3/10/19	8.4.3.1.19	⟨data⟩	⟨data⟩	{wear=pass,...}	@mat3

*Figure 1. A database fragment from $\mathbb{S} =$ **StructEng**. The empty matProps column will be the target of our integration procedure in Section 5.*

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

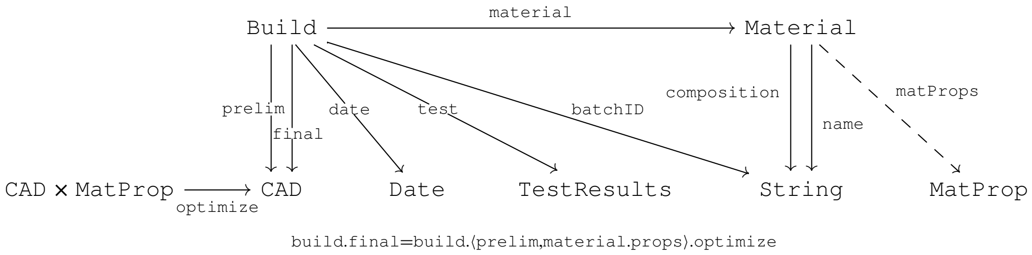

Figure 2. Schema for the model $\mathbb{S} = \textbf{StructEng}$*. The dashed arrow* `matProps`*, however, belongs to a schema extension* $\mathbb{S} \subseteq \mathbb{S}'$*.*

different types of interpretations. Here we use two categories, **Type** and **Set**, to model computations and data, respectively.

Modulo some important technicalities (especially regarding stateful aspects of computation), we can think of a statically-typed programming language like Java or Haskell as a category. Its objects are types and its arrows are methods that takes one type as an input and returns another as output. Composition is defined by calling one method on the output of another. **Set** is similar, except that we exclusively consider the input-output behavior of functions, without regard to the way that values are determined.

We can then use mappings called functors to assign meaning to our schematic elements. For example, the team's implementation of the `optimize` method and other datatypes defines a functor $\textbf{impl} : \mathbb{S}_1 \to \textbf{Type}$, where $\mathbb{S}_1 \subset \mathbb{S}$ is the subschema consisting of types and computations. Similarly, the team's database defines a functor $\textbf{data} : \mathbb{S} \to \textbf{Set}$. Notice that the latter includes the entire schema, not just entities and correspondences, because attributes (data columns) involve types as well as entities.

The final element of our model is a coherence constraint, specifying that the input-output behavior attributed to a method by the **data** functor should agree with the actual behavior computed from **impl**. We can formalize this diagrammatically (see Definition 1) using the fact that running a method and evaluating its outcome defines a third functor $\textbf{eval} : \textbf{Type} \to \textbf{Set}$.

5 MODEL INTEGRATION

In this section we use the models and mappings defined previously to describe a new method and methodology for model integration. In contrast to most discussions, which focus on identifying the minimal merge of two models (ideally automatically), here we emphasize the importance of extending a minimal integration with additional bridging structure to mediate mismatches between distinct but related concepts.

Our integration procedure can be broken down into four steps, as shown in Figure 3. Given two models $\mathbb{S}$ and $\mathbb{T}$ to be integrated, the first step is to identify the overlap $\mathbb{O}$. Next, one constructs the minimal merge $\mathbb{P}$ via pushout. The resulting merged model is then extended with additional bridging structure. Finally, one typically restricts attention to a smaller chunk of the merged and extended model. We illustrate this procedure by integrating the structural engineering model described in Section 4 with the data from the materials engineering team who formulates and tests the raw materials.

Write $\mathbb{S} = \textbf{StructEng}$ for the structural engineering team's model. Recall that $\mathbb{S}$ involves two CAD models, one preliminary and another optimized, and the latter is created by an optimization algorithm which takes a material-properties profile as input. In order to collect the necessary information both now and in the future, the structural engineers would like to interface with the material team to identify nominal values for the necessary properties. The material team's schema and data define a second model $\mathbb{T} = \textbf{MatEng}$.

The materials team creates new material formulations in batches, and tests each batch to record the resulting material properties. A fragment of their model is shown in Figure 4. Our goal in this section is to fill the extended schema from the previous section (i.e., the arrow `matProps` : `Material` → `MatProps`) through reference to the data in $\mathbb{T}$.

Step 1: Match. We begin by identifying overlapping elements (*matches*) which are shared between the two models. We can consider this question at three separate levels: type, schema and data. On the type

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

Input: Models $\mathbb{S}$, $\mathbb{T}$; Schema extension $X : \mathbb{S} \to \mathbb{S}'$.

1) Identify the overlap $\mathbb{S} \xleftarrow{F} \mathbb{O} \xrightarrow{G} \mathbb{T}$

2) Construct a minimal merge (pushout) $\mathbb{P}$

3) Extend with bridging structure $L : \mathbb{P} \to \mathbb{E}$

4) Restrict to a smaller model $\mathbb{S} \xrightarrow{X} \mathbb{S}' \xrightarrow{M} \mathbb{E}$

$$\begin{tikzcd}
(1) & & (2) & (3) & (4) \\
& \mathbb{S} \arrow[rrr, "X"] \arrow[dr, "H"'] & & & \mathbb{S}' \arrow[dl, "M"] \\
\mathbb{O} \arrow[ur, "F"] \arrow[dr, "G"'] & & \mathbb{P} \arrow[r, "L"] & \mathbb{E} & \\
& \mathbb{T} \arrow[ur, "K"'] & & &
\end{tikzcd}$$

Figure 3. Four phases of model integration.

side, we can consult the implementation functor to identify matching types and computations; since model transformation functors are required to commute over implementation, any types in $\mathbb{O}$ must be implemented the same way in $\mathbb{S}$ and in $\mathbb{T}$. In this case, $\mathbb{S}$ and $\mathbb{T}$ share three types, `String`, `Date` and `MatProp`, and no computations.

Unfortunately, contra the vision of Bernstein *et al.* (2000), universal matching operators for ontological and data elements cannot exist. The easiest way to see this is to identify two different semantic situations (corresponding to different integrated models), both of which are compatible with the same component models. For example, semantically distinct data elements may share all the same attribute values, so that they are "observationally equivalent". Usually we correct for this by keying, but keys are model-specific and cannot be used to disambiguate data elements coming from different models. In matching this leads to problems like, e.g., deciding whether `John Smith` from model $\mathbb{A}$ corresponds to the `John Smith` in model $\mathbb{B}$ who lives at `123 Park Avenue`, or the one who lives at `456 Seventh Street`.

Fortunately, there are a variety of heuristic methods that can assist users by prototyping schema and data overlaps. Matching based on names is one obvious strategy: both models contain a `Material` table, suggesting an overlapping element in their schemas. Although names play no role in model transformations—namespaces in different categories are disjoint—common usage can still provide a useful guide for automated prototyping of maps.

Such linguistic matches need not be exact; using typeside computation we can implement arbitrarily complicated similarity measures (e.g., edit distance, word2vec) on `String` and other types to assess possible matches. Conversely, we can*not* assume that matching terminology implies an actual overlap: both $\mathbb{S}$ and $\mathbb{T}$ contain `date` fields (associated with `Build` and `Batch`), but there is no indication that these represent the same information.

As well as matching based on similarities in the namespace of a model, we can also try to identify "correspondences" in the data itself (Bernstein and Melnik, 2007). For example, there is a fairly unique string `"Fe-14Cr-4Ni-4"` which occurs in both `Material` tables. This suggests that the two data elements `@mat1`$\in$ **StructEng** and `@mat3`$\in$ **MatEng** might be matched in the overlap. This is particularly powerful, as it allows us to infer schema matches as well as data; here we can infer that the `composition` attribute in $\mathbb{S}$ corresponds to the `chemistry` attribute in $\mathbb{T}$.

The two approaches are complimentary. When we find namespace matches, we can use them to narrow the search for correspondences. When matches from names and correspondences line up, this provides independent support in our assessment of a match. Though not demonstrated in our example, correspondences can also invalidate potential namespace matches.

We should emphasize that our integration procedure is agnostic with respect to (or better yet, parametric over) the strategy for identifying matches. The problem lacks a canonical solution and different

Batch	@batch4
material:Material	@mat3
date:Date	1/20/19
thermProps:ThermProps	⟨conductivity = ...⟩
testProps:MatProps	⟨elasticity = ...⟩

$\xrightarrow{\texttt{material}}$

Material	@mat3
chem:String	Fe-14Cr-4Ni-4

*Figure 4. A fragment of the model $\mathbb{T}$ = **MatEng** with selected data values.*

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

matching algorithms may be appropriate for different contexts (e.g., domain-specific databases). Consequently, we neither encourage nor discourage any particular matching algorithm, instead providing a precise and flexible context for defining the problem and its potential solutions.

Step 2: Merge. Next we use the diagram constructed in Step 1 to build the pushout $\mathbb{P} = \mathbb{S} \underset{\mathbb{O}}{\oplus} \mathbb{T}$, which represents the minimal merge of $\mathbb{S}$ and $\mathbb{T}$ relative to the specified overlap $\mathbb{O}$. The merged model comes equipped with transformations $\mathbb{S} \xrightarrow{P} \mathbb{P} \xleftarrow{Q} \mathbb{T}$ mapping the component models into the merge. The merged model consists of a copy of $\mathbb{S}$ and a copy of $\mathbb{T}$ side by side, except that those objects and arrows which appear in the overlap $\mathbb{O}$ are shared between the two copies.

Here we can see an example of the use of labeled nulls. Suppose the **MatEng** model also contains a brass alloy: $\langle @\texttt{mat7} : \texttt{chemistry} = \texttt{Cu} - \texttt{7Zn} - 3 \rangle$. This has no match in the **StructEng** model, so the the integrated database has no `name` attribute for that entry. Rather than leaving such holes blank, the Σ and colimit operations create labeled variables to track their provenance (e.g., `@mat7.name`). Sometimes we can infer missing information from these constraints: if we formalized the intuitive naming scheme in the **StructEng** model we could deduce that `@mat7.name="CopperZinc"`.

Step 3: Extend. In the third step of the integration we extend $\mathbb{P}$ with additional bridging structure. This allows us to introduce new model elements which do not occur in either component but which are needed to interpolate between them.

Recall that our goal is to identify the nominal material properties for each material in the database. However, the materials engineers associate material properties with batches, not with materials, and we will need to aggregate these empirical values in order to obtain the desired nominal quantities. We proceed in three steps.

First we observe that any input-output function $\texttt{f} : \texttt{X} \to \texttt{Y}$ can be converted into an auxiliary function $\texttt{Fiber}_\texttt{f} : \texttt{Y} \to \texttt{List}[\texttt{X}]$. Here $\texttt{Fiber}_\texttt{f}(\texttt{y})$ is a list of all the elements $\texttt{x} \in \texttt{X}$ such that $\texttt{f}(\texttt{x}) = \texttt{y}$ (called the inverse image or the fiber of `f` over `y`). In particular, the $\mathbb{T}$-arrow $\texttt{material} : \texttt{Batch} \to \texttt{Material}$ generates an associated mapping $\texttt{Material} \to \texttt{List}[\texttt{Batch}]$ which collects together all the batches of each material.

Next we can apply the `testProps` arrow to each batch, one by one, to generate a list of `MatProps`. In computer science, especially functional programming, this is refered to as a "map" operation. More specifically, any function $\texttt{f} : \texttt{X} \to \texttt{Y}$ defines an associated function $\texttt{map}(\texttt{f})$ which acts on lists in an obvious way, sending $\langle \texttt{x}_\texttt{i} \rangle \mapsto \big\langle \texttt{f}(\texttt{x}_\texttt{i}) \big\rangle$.

Finally, we need a way of collapsing this collection of material profiles down to a single nominal value. We model this as a computation $\texttt{List}[\texttt{MatProp}] \to \texttt{MatProp}$. If a `MatProp` element is just a vector of properties, then this computation could be be implemented by simply averaging each component. In other circumstances we may need a more sophisticated algorithm to aggregate these values. For example, destructive testing may mean that we do not have all property values for each `MatProp` element so that a simple average is not appropriate. Alternatively, we may want to factors in the uncertainties associated with of different measured values, giving greater weight to more precise observations.

In total, then, we add three new arrows into the extension model $\mathbb{E}$, in addition to some constraints associated with the `Fiber` construction:

$$\begin{aligned}
&\texttt{batches} : \texttt{Material} \to \texttt{List}[\texttt{Batch}]\\
&\texttt{map}(\texttt{testProps}) : \texttt{List}[\texttt{Batch}] \to \texttt{List}[\texttt{MatProp}]\\
&\texttt{aggregate} : \texttt{List}[\texttt{MatProp}] \to \texttt{MatProp}
\end{aligned}$$

Step 4: Restrict. In the final step of the integration we clean up our model. The integration $\mathbb{E}$ contains a lot of extraneous material about batches and thermal properties in addition to the `MatProp` attribute and `matProps` arrow that we want. We can amend this by projecting to a smaller schema.

Specifically, we create a functor $M : \mathbb{S}' \to \mathbb{E}$ in order to extract the instance data we want, where $\mathbb{S}' \supseteq \mathbb{S}$ is the extension schema defined in the previous section. In order to define M, we first observe that $\mathbb{S}$ already sits inside the pushout $\mathbb{P}$, and hence inside $\mathbb{E}$ as well.

Consequently, the only element of the M-mapping that must be defined is the image of the new attribute $\texttt{matProps} : \texttt{Material} \to \texttt{MatProps}$. The choice is obvious given our work building $\mathbb{E}$; we need to send `matProp` to the composite `batches.map(testProps).aggregate`.

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

Finally, we can use the mapping M to pull data from the integrated, extended schema $\mathbb{E}$ back to the target $\mathbb{S}'$. This black-boxes the manipulations used to construct the `matProps` values, storing only the desired data values as attributes.

There are a few points worth noting before we close.

First, the procedure described above is best regarded as a rational reconstruction of the data integration process, and is pragmatically naïve in a number of ways. For example, our model makes little provision for typos and other data value mismatches; if we (correctly) match two data elements and their `address` attributes, but one contains `"100 Bureau Dr."` while the other has `"100 Bureau Drive"`, we are left with the unsavory consequence that `"Dr."="Drive"` *as strings*. Our model can handle this (remove `address` from the overlap and introduce a disambiguation function in the bridging structure), but in a practical setting this sort of workaround should be handled automatically. At the same time, our reconstruction can be quite valuable in setting up such heuristics and recognizing when they are necessary.

Second, our approach provides the foundation for a distributed, bottom-up approach to shared semantic representations. Although our example focused on a binary integration problem, the colimit-based method generalizes immediately to n-component integration. This could allow for the creation of domain-specific libraries which can be mixed together to build new models and applications.

In our example, the `chemistry/composition` attributes are semantically deficient, using a `String` attribute to represent what is obviously some sort of more structured data. It would be better to design a new computational type to store this information, with explicit hooks to related concepts like `Element` and `chemicalComposition`. Designing such a package may be quite complicated, and is rarely worth the effort for a single user, but could be quite valuable if shared across a community. By encoding domain knowledge into our models, we make future integrations and extensions easier; for example, a better model of chemical composition would make it easier to analyze differences between iron- and nickel-based alloys.

Finally, we note that our methodology can be applied incrementally to evolve and improve models over time. A careful reader may have recognized another correspondence in our example, between the `batchID` associated with a `Build` and the `date` associated with a `Batch`. This is unlikely to represent a direct match between objects (multiple builds refer to the same `batchID`), but does indicate the presence of a bridging arrow `Build` → `Batch` in the extension $\mathbb{E}$. We didn't need this information today so we were free to ignore it, but if tomorrow we need to track the provenance of our material to respond to a product recall we can build on the work we have already done to pull in this information as well.

CONCLUSION

This paper has developed some introductory steps in the formal foundation of MM, and opens up a range of research questions both theoretical and practical.

Our models involve several parameters which could be tuned to provide greater flexibility. For example, the base category $\mathbb{2}$ provided an abstraction barrier between entities and types, and could be generalized to allow for more sophisticated abstractions.

Other interesting questions concern the relationship between bridging structure and other stages of the integration. For simplicity, we included a strong requirement of type-side commutativity in our mappings, but this could be weakened to a natural transformation, incorporating type-casting directly into our model transformations. Similarly, weighted colimits are a higher-dimensional generalization of ordinary colimits, and could be used to incorporate some types of bridging structure into the overlap between models.

Practically speaking, there is much more to MM than just integration, and we can further validate our models and methodology by applying them to problems like documentation and traceability, data cleaning and transformation and the incorporation of tools and workflow, just to name a few. Future work will test our framework against the model management desiderata outlined in Bernstein and Melnik (2007).

Another important topic is implementation. In fact, much of the discussion here can already be implemented in the Algebraic Query Language; see (removed for review) for an earlier version of our model integration method. Further progress will require a careful consideration of the runtime structures which manage and modify our models. It would also be quite useful to extend our results to more expressive schemas, perhaps through reference to structures in the **Type** category (e.g., coproducts, function types).

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press

REFERENCES

Adámek, J. and Rosicky, J. (1994), *Locally presentable and accessible categories*, Vol. 189, Cambridge University Press.

Alagić, S. and Bernstein, P. A. (2001), A model theory for generic schema management, *in* "International Workshop on Database Programming Languages", Springer, pp. 228–246.

Awodey, S. (2010), *Category theory*, Oxford University Press.

Bernstein, P. A., Halevy, A. Y. and Pottinger, R. A. (2000), "A vision for management of complex models", *ACM Sigmod Record*, Vol. 29 No. 4, pp. 55–63.

Bernstein, P. A. and Melnik, S. (2007), Model management 2.0: manipulating richer mappings, *in* "Proceedings of the 2007 ACM SIGMOD international conference on Management of data", ACM, pp. 1–12.

Chandrasegaran, S. K., Ramani, K., Sriram, R. D., Horváth, I., Bernard, A., Harik, R. F. and Gao, W. (2013), "The evolution, challenges, and future of knowledge representation in product design systems", *Computer-aided design* Vol. 45 No. 2, pp. 204–228.

Eckert, C. M., Wynn, D. C., Maier, J. F., Albers, A., Bursac, N., Chen, H. L. X., Clarkson, P. J., Gericke, K., Gladysz, B. and Shapiro, D. (2017), "On the integration of product and process models in engineering design", *Design Science* **3**.

Johnson, M., Rosebrugh, R. and Wood, R. (2002), "Entity-relationship-attribute designs and sketches", *Theory and Applications of Categories* Vol. 10 No. 3, pp. 94–112.

Levy, S., Subrahmanian, E., Konda, S., Coyne, R., Westerberg, A. and Reich, Y. (1993), An overview of the *n*-dim environment, Technical report, Engineering Design Research Center, Carnegie-Mellon University.

Männistö, T. and Sulonen, R. (1999), "Evolution of schema and individuals of configurable products", in *International Conference on Conceptual Modeling*, Springer, pp. 12–23.

Microsoft Research (2001–2011), "Model management". http://www.microsoft.com/en-us/research/project/model-management/.

Patterson, E. (2017), "Knowledge representation in bicategories of relations", *arXiv preprint arXiv:1706.00526*.

Piela, P. C., Epperly, T., Westerberg, K. and Westerberg, A. W. (1991), "ASCEND: An object-oriented computer environment for modeling and analysis: The modeling language", *Computers & chemical engineering* Vol. 15 No. 1, pp. 53–72.

Rosebrugh, R. and Wood, R. (1992), "Relational databases and indexed categories", in *Proceedings of the International Category Theory Meeting 1991, CMS Conference Proceedings*, Vol. 13, pp. 391–407.

Sarala, P., Breiner, S., Subrahmanian, E. and Sriram, R. (2018), Deconstructing UML, part 1: The class diagram. Under review.

Schultz, P., Spivak, D. I. and Wisnesky, R. (2016), "Algebraic model management: A survey", in *International Workshop on Algebraic Development Techniques*, Springer, pp. 56–69.

Spivak, D. I. (2012), "Functorial data migration", *Information and Computation* Vol. 217, pp. 31–51.

Subrahmanian, E., Reich, Y., Konda, S., Dutoit, A., Cunningham, D., Patrick, R., Thomas, M. and Westerberg, A. W. (1997), "The n-dim approach to creating design support systems", in *Proc. of ASME Design Technical Conf.*

Wynn, D. C., Nair, S. M., Clarkson, P. J. *et al.* (2009), "The p3 platform: An approach and software system for developing diagrammatic model-based methods in design research", in *DS 58-1: Proceedings of ICED 09, the 17th International Conference on Engineering Design, Vol. 1, Design Processes*, Palo Alto, CA, USA, 24.-27.08. 2009, pp. 559–570.

ACKNOWLEDGMENTS

We are grateful to David Spivak and Ryan Wisnesky for many discussions on and around these topics. Blake Pollard was supported by NSF grant # 1746077.

Disclaimer: This paper includes contributions from the U. S. National Institute of Standards and Technology, and is not subject to copyright in the United States. Commercial products are identified in this article to adequately specify the material. This does not imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply the materials identified are necessarily the best available for the purpose.

https://doi.org/10.1017/dsi.2019.202 Published online by Cambridge University Press