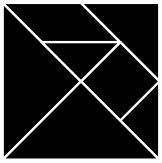
GSDLAB TECHNICAL REPORT

Towards Category Theory Foundations for Model Management

Zinovy Diskin, Tom Maibaum, Krzysztof Czarnecki

GSDLAB–TR 2014-03-03 April 2014



Generative Software
Development Lab

Generative Software Development Laboratory
University of Waterloo
200 University Avenue West, Waterloo, Ontario, Canada N2L 3G1

WWW page: http://gsd.uwaterloo.ca/

Towards Category Theory Foundations for Model Management

Zinovy Diskin[1,2], Tom Maibaum[1], and Krzysztof Czarnecki[2]

[1] NECSIS, McMaster University, Canada
{diskinz|maibaum}@mcmaster.ca
[2] University of Waterloo, Canada
{zdiskin|kczarnec}@gsd.uwaterloo.ca

Abstract. The paper aims to demonstrate that category theory (CT) methods are appropriate in the model management context. We show that CT *naturally* appears on stage, when we model such basic constructs as model merge and parallel composition accurately and consistently. We present categorical constructs in a tutorial-like style by applying them to simple labeled transition systems (LTSs), and reveal new technical aspects of merging and composing LTSs.

1 Introduction

Model management (MMt) is suffering from the structural mismatches and uncertainties about its methods and tools, often designed without proper semantic support. An integrating semantic foundation is badly needed, but its development is challenging. The extreme diversity of models, relationships between them, and operations over them seems to resist attempts at discovering common mathematical patterns underlying various MMt scenarios. Category theory (CT), well known for its unifying power as *the* comparative mathematics of structure, is a good candidate for the job. It has already been applied in the field for modeling basic MDE notions such as intermodeling, mega- and meta-modeling, and model transformation and synchronization (references will be provided shortly). However, categorical methods are often considered to be excessively abstract and complicated, and categorical modeling is often thought of as modeling for the sake of modeling rather than being really beneficial for the subject matter.

We understand the roots of such an opinion, but believe it is not constructive for building semantic foundations, and even for practical MMt. We see the role of CT for MMt as being similar (perhaps, very similar, cf. [4,12]) to the role of the relational theory for data management. The mathematical underpinning of relational theory is also very non-trivial and requires multiple formal constructs and details. Nevertheless, the database community has successfully developed an engineering approximation of the theory, and built a whole industry based on it. Relational thinking is now considered to be absolutely natural, and basic, for data management, although it was not so at the time of record-based data processing, when the relational model emerged and dramatically changed the

field. We believe that with a suitable approach, a similar path can be followed for MMt.

We see two major directions to pursue. One is developing technical applications of CT for particular MMt tasks, as done in the papers cited in the next section. The other is to explore the very foundations of the MMt-to-CT (and back!) correspondence: what ideas and constructs can be seen as shared by the two disciplines, and what aspects are private (a.k.a. implementation details and formal hairsplitting). Seen in the human-centric perspective, these issues give rise to several important questions. How much naturally can categorical models be presented to an MMt practitioner? Are categorical diagrams doomed to be perceived by him as an artificial mathematical imposition over his simple MMt sketches on the whiteboard? Or as a formal dress for his pretty MMt code (good for writing papers, but otherwise useless)? Usability issues are key for acceptance of any mathemtical framework, and addressing them needs a wide range of activities ranging from the development of practical approximations and interfaces to fundamental theories (e.g., analogous to SQL), tool support, and empirical studies, to vindicate (or not!) the proposed approaches, and last but not least, effective education materials (e.g., tutorials and textbooks) are crucial.

The present paper is a small step within the large program above, and its goals are two-fold. First, we explore how natural categorical modeling is for the MMt field. A fundamental question we address is whether the peculiarities of categorical models are foreign to the MMt scenarios they model (and imposed by the modeling language, i.e., CT), or do these peculiarities reflect the corresponding features of the scenario at hand. If the former is true, CT is not suitable and the story is over. If the latter holds (which we hope to demonstrate), the story goes on with the development of the theory and the usability issues identified above. We will show that categorical *arrow encapsulation* is a foundational idea that can do the job. Second, we tried to write the paper as a mini-tutorial on principles of CT-modeling, which an MMt practitioner could read and think about, then, perhaps, take a look at the long version of the paper [13] and some of the references, and try to apply some of the ideas in her everyday work.

To achieve these goals, we will take classical behavior models — labeled transition systems (LTSs) as sample models, consider their merge and parallel composition as sample MMt scenarios, and build their categorical models paying attention to the basic ideas and exploring the questions mentioned above. As a byproduct, we present several technical details of LTSs and operations over them that appear to be novel. In particular, we show several benefits of considering LTSs as labeled categories rather than labeled graphs.

Our plan in the paper is as follows. In the next section we discuss the related work. In Section 3, we demonstrate principles of categorical modeling by considering matching and merging of LTSs . Section 4 provides several formal definitions, and argues that an LTS should be a labeled category. We also discuss how far the constructs we introduced with simple examples can be extended beyond them. Section 5 presents the parallel composition of LTSs as a construct dual to merge. In Section 6 we summarize our findings and conclude the paper.

2 Related work

A general call to employ CT in MDE was made by Don Batory in his invited lecture for Models 2008 [3]. The categorical language was used informally, and in a very general context: only the very basic categorical ingredient, the arrow composition, was mentioned. Applying CT for formal modeling of several basic MDE constructs, intermodeling [6,14,8], mega-modeling [12], metamodeling [26,24], and model transformation and synchronization [7,25,9,28] (this area is especially amenable to a categorical treatment based on the triple graph grammars (TGG) [20,18,1,22,21]), revealed a much richer structural landscape, and required much more advanced categorical tools (institutions, sketches, monads, Kleisli mappings, adhesive categories, fibrations). Results presented in these, and many other papers not mentioned, can be seen as evidence of CT's applicability to building mathematical models for MDE. Our goals in the presents paper are different (although related): we aim to show several basic principles of categorical modeling and discuss them in the (meta)context outlined in the introduction.

Several remarks about our sample MMt scenarios with LTS: An LTS is a classical behaviour model defined as a ternary transition relation $T \subset S \times L \times S$ with a set of states S and a set of labels L (a.k.a. an alphabet). Our chain of definitions making an LTS a morphism $\lambda\colon T \to L$ of, consecutively, graphs, precategories, and finally categories, seems to be novel. We are not aware of viewing commutativity between sequences of transitions as a major behavior modeling construct.

Using colimits for modeling various operations of "putting widgets together" can be traced back to Goguen's pioneering work [17], and has often been used in computer science, databases, ontology engineering and software engineering; several annotated references relevant to MDE can be found in [14]. Using limits for synchronized parallel composition is less well known [19]; the closest to ours setting is, probably, in [16], where they use limits for composition of alphabets seen as (pointed) sets. But for us, an alphabet is a category, an LTS is a functor into this category, and we compose both alphabets (as types) and their "instances" (states and transitions).

3 Intermodeling, Model Merge, and Colimit

We say *model merge* to refer to the following MDE scenario. Several models expressing *local* views of the system are first *matched* by linking elements of the models that correspond to the same system element. Then models are integrated into a single *global* model (either physically or logically), which includes all data from the local models but without redundancy, in accordance with the match.

Our goal is to show that modeling model merge (an MDE task) by the *colimit* operation (a mathematical construct) is an unexpectedly accurate mathematical model of the phenomenon.

3.1 Simple Match and Merge

Two independent consultants, Ben and Bill, have designed two consumption models. Ben's model ($M1$ in Fig. 1) states that buying an apple is a wise way

of spending your dollar, but first you need to work and earn it. Bill agrees with Ben about the necessity to work first, but suggests spending the dollar earned on buying a cake (model $M2$).

Suppose Ben and Bill want to merge their models into an integrated model U without data redundancy and loss. They specify correspondences between the models by bidirectional links ℓ_x ($x = 0, \$, w$) connecting (*matching*) elements considered to be "the same". Merge is easily performed "by hand" resulting in model U in Fig. 1. What Ben and Bill did was, first, disjoint merging of the two graphs and, then, gluing together matched elements.

However, designing tools for automatic merge for models with thousands of elements, and complex intermodel relationships, would need a deeper, and formal, understanding of the merge procedure. Let us see how the categorical analysis could help.

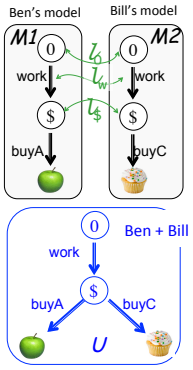


Fig. 1. Model-based merge (case 1)

The first prescription of CT is to take relationships seriously and explicitly specify them and their properties. We first notice that three intermodel links ℓ_x ($x = 0, \$, w$) respect the structure of the modes: they relate states to states, transitions to transition, and source and target of related transitions are respected as well. To make this structure preservation explicit, it is convenient to reify every bidirectional link $\ell_x : e1_x \leftrightarrow e2_x$ as a special relationship object, $\ell_x^\bullet$, and then combine these into a model R endowed with mappings r_1, r_2 to restore the end elements of the links (see the upper part of Fig. 2 where object $\ell_x^\bullet$ is labeled by x). Model R is also an LTS, and mappings r_1, r_2 respect the LTS-structure as explained above. We will say that the mappings are *structure preserving*, and call them *LTS-morphisms*. A pair of mapings $r = (r_1, r_2)$ with a common source R is called *a span*, model R is its *head*, and mappings r_i are the *legs*. We will often write a span as a triple $r = (r_1, R, r_2)$, in which the head is explicit, and sometimes denote the entire span by R, if the legs are clear form the context.

Target and source multiplicities are important properties of mappings. By default, we will assume all mapping to be totally-defined and single-valued, and their target multiplicity [1] is not shown. The source multiplicity will vary, and in the case we consider, it is [0..1] for both mappings r_i (i.e., the mappings are not *surjective* or not *covering*). So, each of the two models has a shared part (the image of mapping r_i), and a private part (the rest of the model).

Thus, matching results in a span, serving as the input data for an algebraic operation of *pure merge*. In Fig. 2, the input models and mappings are shaded, while the output ones are blank (and blue with a color display: the blue color is meant to recall a *mechanical* computation). We begin our discussion of the operation by specifying what exactly is produced by the merge.

First of all, note that each of models Mi ($i = 1, 2$) can be mapped to the merged model via a totally defined and single-valued mapping u_i, which is an LTS-morphisms. Totality of u_i ensures that the merge is *lossless*: no data of

the original models are lost. Single-valuedness says that merging does not split elements, which is a reasonable requirement too.

In the case we consider, mappings u_i are injective (no two elements are glued together), but below we will see another merge case, in which gluing occurs and is a desirable property. As model M_1 has a private part, mapping u_2 is not surjective, and similarly non-surjectivity of r_2 implies non-surjectivity of u_1. (Strong connections between properties of mappings $r-1$ and u_2, and r_2 and u_1 is provable for the algebraic operation of colimit.) However, the pair of mappings (u_1, u_2) is *jointly surjective* or *jointly covering*: for each element $e \in U$ there is either element $e_1 \in M_1$ s.t. $u_1(e_1) = e$, or $e_2 \in M_2$ s.t. $e = u_2(e_2)$, or both. We will often skip "jointly" and say that the *cospan* (u_1, u_2) is *covering*. Thus, equality $U = u_1(M_1) \cup u_2(M_2)$ is yet another important property of the merge operation often stated as the *No-junk* property: the merge model contains nothing that is not contained in the local models.

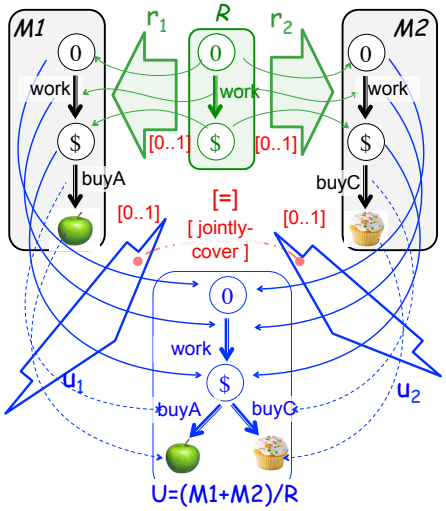


Fig. 2. Map-based merge (case 1)

Finally, for any element $e \in U$ of the merge, $e \in u_1(M_1) \cap u_2(M_2)$ iff there is some $\ell^{\bullet}_e \in R$ such that $\ell^{\bullet}_e.r_1.u_1 = e = \ell^{\bullet}_e.r_2.u_2$. Thus, $r_1; u_1 = r_2; u_2$ where ; denotes mapping composition. We say that the square formed by these four mappings is *commutative*, and label the diagram with symbol [=]. Note a direct visual representation of commutativity: cycles formed by links constituting the mappings are closed. Later we show that for a given span $r = (r_1, r_2)$ between models $M1$, $M2$, there is a unique (up to isomorphism) covering cospan $u = (u_1, u_2)$ that completes the span r up to a commutative square, as shown in Fig. 2. We denote the head of this cospan by $(M1 + M2)/R$: indeed, the more links are contained in mappings r_i, the more elements in the disjoint union are glued together, and the smaller is model U. This operation, and its result, are called *colimit*; cospan U is the colimit of span R.

Note that the up-to-isomorphism nature of the definition of colimit, often considered as an unnecessary complication imported by CT, is exactly adequate for practice: object IDs of the merged model elements depend on the merge tool (or even the state of the tool) rather than being uniquely defined by the input data. Practical merge is defined up to isomorphism exactly like the colimit is.

Now compare Fig. 1 and Fig. 2. The latter specifies several important properties of merge via properties of the mappings involved. The former is a loose specification with all these properties left implicit. This looseness quickly accumulates when we merge several models interrelated by a complex system of spans. Consider, for examples, two new models $M3$, $M4$ at the left top corner of Fig. 3,

which together with the match span R_{34} specify the convergence of eating an apple or a cake to a happiness state. (Here we used a bidirectional block-arrow containing a single bidirectional link as a shorthand for the respective span, whose head has an only object.) If we want to merge all models without redundancy, we also need to match apples in $M1$ and $M3$, and cakes in $M2$ and $M4$. The entire system of models and matches is shown in the upper half of Fig. 3, and the result of merge is in the lower half. Note four commutativity conditions respected by the merge (corresponding to the four input spans). More complex examples can be found in [27,14].

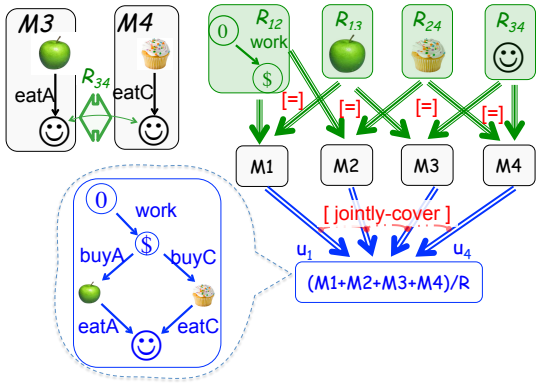


Fig. 3. Multimerge

In the simple case we consider, merge can be easily done by hand, but for industrial size models and matches, removal of mappings and their properties from the specification, especially multiple commutativity requirements, would make coding very error-prone. And if even a merge tool conforming to a loose specification without mappings works properly, important requirements would be hidden in code, which would then be difficult to analyse, test, and maintain.

Our next goal is to analyse whether simple patterns considered above work for more complex cases of inter-model relationships.

3.2 Complex Match and Merge, I: Pseudo-one-to-many Links

In MDE practice, intermodel relationships are often more complex than the one-to-one matches considered so far. A simple *one-to-many* match is shown in Fig. 4(a). Model M_1 says you can convert a dollar into a smile by either buying and eating an apple, or by buying and eating a cake. Model M_2 is more abstract and says you can convert a dollar into a smile by either *healthyLife* or *happyLife*, but is not specific about details of what should really be done. Suppose we know that by *healthyLife* model M_2 actually means buying and eating an apple, so that transition *healthyLife* matches two transitions in M_1.

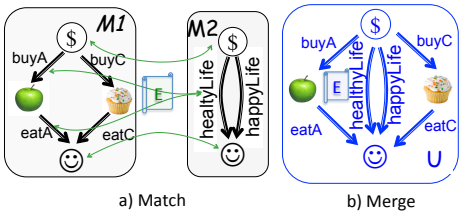


Fig. 4. Model-based merge (case 2)

Also, the pair of matching links is labeled by an expression E:: $healthyLife = buyA; eatA$ specifying details of the

one-to-many relationship. The merge can easily be produced by hand Fig. 4(b), but building a general algorithm in this case is even more complicated than for one-to-one matching. Indeed, now the merge algorithm needs to manage expressions of, generally speaking, different kinds. Expression management was declared as one of the big problems of model management in [5].

Let us see how the case is treated categorically. A key observation is that a one-to-many relationship is replaced by a one-to-one relationship to a derived transition *buyA*; *eatA*, as shown in Fig. 5, in which the bidirectional horizontal arrow is again used as a shorthand for the respective span — this syntactic sugar is intuitive and makes diagrams more compact. The derived transition in model M_1 is shown by a dashed arrow (blue with a color display), and the respective triangle diagram is marked with symbol [;] specifying the operation of arrow composition.

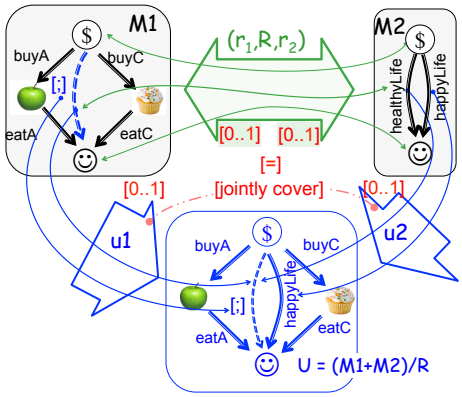


Fig. 5. Map-based merge (case 2)

Use of derived transitions allows us to reduce *one-to-many* to *one-to-one* matching, then the simple merge algorithm described above can be directly applied and produces the result shown in the lower part of the figure. The main merge principle of copying all data from original models into the merge is directly realized, and the operation label [;] is copied to U as well and declares the dashed transition as derived by the arrow composition exactly as it was in model M_1. Note we do not need to copy transition *healthyLife* from M_2 to the merge model because it can be derived, and mapping u_2 is still total. In contrast, the naive merge in Fig. 4(b) is corrupted by inclusion of this transition.

The same idea is applicable for other operations on transitions, e.g., their parallel composition, or, in general, any operation over a model's elements. Hence, we consider the same merge operation, but in different categories: of graphs, of graphs with sequential arrow composition, graphs with parallel arrow composition, etc. (a precise categorical framework for this approach is based on the notion of *Kleisli category* discussed in the MDE context in [8], and in the database context in [11]).

3.3 Complex Match and Merge, II: Real One-to-many links.

Now consider yet another case of model overlap in Fig. 6. Suppose that the *happyLife*-transition in model M_2 can mean either buying and eating an apple, or, perhaps, buying and eating a cake. The respective overlap is specified by span R (note that r_2 maps two transition to the same target) , and the merge according to the colimit algorithm is given by cospan (u_1, U, u_2). The inset figure below shows the merge model with derived transitions omitted (they can be always computed if needed), but the commutativity constraint made explicit as it is not

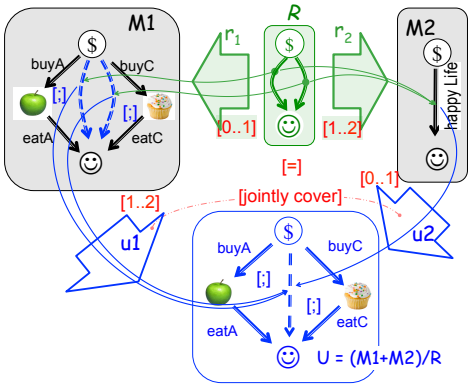
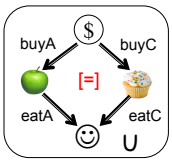


Fig. 6. Map-based merge (case 3)

At first sight it may seem that the colimit-based merge is incorrect as two different transitions in model M_1 are glued together in the merge U, and hence the [=]-constraint is to be declared. Let us consider the case in more detail. The match says that transition *happyLife* is "equal" to *buyA;eatA* and to *buyC;eatC* as well. It implies that *buyA;eatA=buyC;eatC*, which is exactly the commutativity constraint declared in U. Thus, a constraint to model M_1 was actually declared externally by the match rather than by the model itself. This is a typically categorical phenomenon, when some properties of an object are only revealed when this object is related to other objects. As model merge should preserve all input information, the constraint about M_1 stated by the input span is to be respected in the merge model U, and this is exactly what the colimit does. Note how accurately this situation is treated categorically: neither of the original models is changed, but everything needed is captured by a properly specified intermodel span.

4 LTSs as mappings: From graphs to categories



In this section we make the constructions described above precise. We will consecutively define LTSs as labeled graphs, labeled pre-categories, and finally, labeled categories, and motivate why graphs and even precategories are not good enough. We define colimit for labeled categories, thus providing formal support for our MMt activity in Section 3. Some details are omitted to save space, but can be found in the long version [13]; yet we have kept enough formalities to give the reader a feeling of what equation chasing is (and appreciate the power and beauty of CT, which frees its users from tedious transformations).

4.1 LTSs as graph morphisms

We give a precise definition of graphs and morphisms to fix our notation.

Definition 1 (Graphs). A *(directed) graph* G comprises a set $G^\bullet$ of *nodes*, a set $\overrightarrow{G}$ of *arrows*, and two functions, $\mathsf{so}\colon \overrightarrow{G} \to G^\bullet$ and $\mathsf{ta}\colon \overrightarrow{G} \to G^\bullet$. We write $a\colon N_1 \to N_2$ if $N_1 = \mathsf{so}(a)$ and $\mathsf{ta}(a) = N_2$. We will often write $e \in G$ to day that $e \in G^\bullet \cup \overrightarrow{G}$ is an element of graph G.

A *graph morphism* $f\colon G \to G'$ is a pair of functions, $f^\bullet\colon G^\bullet \to G'^\bullet$ and $\overrightarrow{f}\colon \overrightarrow{G} \to \overrightarrow{G'}$, such that the incidence between nodes and arrows is preserved: for any arrow $a\colon N_1 \to N_2$ in G, we have $\mathsf{so}'(\overrightarrow{f}(a)) = f^\bullet(\mathsf{so}(a))$ and $\mathsf{ta}'(\overrightarrow{f}(a)) = f^\bullet(\mathsf{ta}(a))$. □

A classical LTS is a graph T whose nodes are called *states* and arrows are *transitions*; the latter are labeled via a function $\lambda\colon \overrightarrow{T} \to L$ into a predefined set L of *(action) labels*. Thinking categorically, mapping a graph to a set is not a natural construct. To fix it, we assume that both transitions and states have labels, which form a graph L. Indeed, as a rule, applying a transition to a state requires the latter to satisfy some pre-conditions, and the result of transition execution satisfies some post-conditions. These pre- and post-conditions can be encoded by *state labels*, and an LTS thus becomes a triple $M = (T, \lambda, L)$ with T and L graphs and $\lambda\colon T \to L$ a graph morphism. For example, the graph of labels for model $M1$ in Fig. 4(a) could consist of three nodes \$, *snack*, and ☺, and two consecutive arrows *buy*: \$ → *snack* and *eat*: *snack* →☺.

Then an accurate specification of model $M1$ is shown in Fig. 7, where names before colons refer to states and transitions, while names after colons refer to their labels. The latter can be considered as types, and the former as their instances. Similarly, for model $M2$, the graph of labels could be taken to be a single arrow *life*: \$ →☺, while the transition graph has two arrows *healthy* and *happy*.

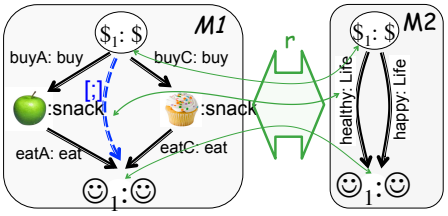


Fig. 7. Labeling

Note that the classical LTS notion is subsumed if we require the set $L^\bullet$ of state labels to be a singleton: then there is only one state label, and a transition can be always applied to a state.

Since an LTS becomes a two-layer graph, an LTS morphism should have two-layers too. The upper box in Fig. 8 illustrates the definition (where $G^{\rightarrow}$ stands for $\overrightarrow{G}$ for typographical reasons). The innermost nodes in the diagram are sets, and thin arrows between them are functions (the diagonal ones are the source and the target functions in the respective graphs). Block nodes and arrows denote systems of sets and functions. The entire diagram specifies an *LTS morphism* $m\colon M \to M'$ as a pair of graph morphisms, $m_T\colon T \to T'$ and $m_L\colon L \to L'$, which commute with labeling, $\lambda; m_L = m_T; \lambda'$ (note symbol [=] in the center of the square diagram).

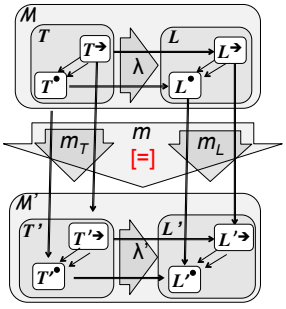


Fig. 8. An LTS morphism

Thus, our diagrams in section 2 were not quite accurate by leaving labels implicit; we employed the fact that labeling was bijective. An accurate description would be to leave diagrams as is, but provide the possibility to zoom-in and reveal the two-layer picture shown in Fig. 8. Correspondingly, merge of LTSs is also a two-layer operation that begins with a two-layer match followed by a two-layer colimit. We will describe them later after we refine our notion of LTS.

4.2 LTSs as functors

Our examples showed that for specifying LTS matching we need sequential composition of transitions, and their labels as well. An important condition to make sequential composition well-behaved is associativity: for any triple of consecutive arrows a,b,c, we have $(a;b);c = a;(b;c)$. This law holds in a majority of the practical interpretations of LTSs we can think of.

Definition 2 (Precategories and prefunctors.). A *precategory* is a directed graph with an associative arrow composition. That is, for any pair (a,b) of consecutive arrows, an arrow $a;b$ is defined, and the associativity law holds.

A *morphism of pre-categories*, or a *pre-functor*, $f\colon C_1 \to C_2$, is a graph morphism that preserves composition: $f(a;b) = f(a);f(b)$ for any pair of consecutive arrows a, b in C_1. □

Now we define an LTS to be a triple $M = (T, \lambda, L)$ with T and L precategories and λ a prefunctor. For example, for model $M1$ in Fig. 7, $\lambda(buyA;eatA) = buy;eat = \lambda(buyC;eatC)$. The diagram in Fig. 8 defining an LTS moprhism m is still valid, but now we interpret the four inner block-arrows as prefunctors.

Figure 7 specifies a span of LTS morphisms in a briefed notation with bidirectional links instead of two leg. Intermodel links are assumed to have two layers and thus encode two spans: between transitions and between labels. For example, the middle link says that transitions *buyA;eatA* and *healthy* are matched over respectively matched labels *buy;eat* and *life*. Thus, the head of the span is a (two-layered) LTS and its legs are (two-layered) morphisms.

Definition 3 (Categories). A *category* is a precategory with a special loop arrow id_X (called *identity*) assigned to each node X. The following equations are to hold for any arrow $a\colon X \to Y$: $\mathsf{id}_X;a = a = a;\mathsf{id}_Y$.

A morphism of categories, or a *functor*, $f\colon C_1 \to C_2$, is a pre-functor that sends identities to identities: $f(\mathsf{id}_X) = \mathsf{id}_{f(X)}$ for any node $X \in C_1^\bullet$. □

Do we really need categories, or is the notion of pre-category good enough for our modeling goals? Consider yet another consumer-oriented model M_c in Fig. 9 (where some of the composed labels are omitted). The model says that after two cycles of working and buying, the system needs to take a rest to return to the initial state. To formalize this condition, we introduce an *idle* loop id_{0_1} at state 0_1 (of type id_0), and postulate equality *w1;b1;w2;b2;r*=id_{0_1} that holds along with the respective label equality *work;buy;work;buy;rest*=id_0 (note the equality symbol between the two transitions). Similarly, the equality symbol near state $\$_1$ says that *b1;w2*=$\mathsf{id}_{\$_1}$: *working* results in the same amount of money (at least, an amount affording the next *buying*, which is considered to be the same state).

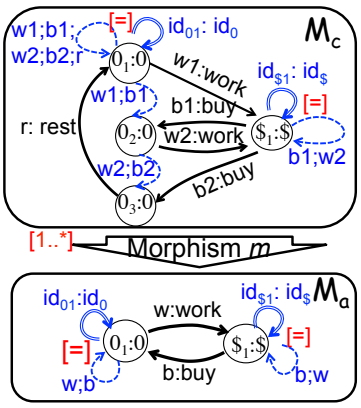


Fig. 9. Idle actions

Model M_a in Fig. 9 is a more abstract view of the same behavior. In this view, the difference between states 0_1, 0_2, and 0_3 is ignored, each cycle *work;buy*

returns to the same state, and *rest* is not needed. The relationship between two models is specified by an LTS morphism m: $M_c \to M_a$ (see Fig. 8). The transition (instance) part of the morphism maps all "vertical" arrows in M_c, the loop $w1; b1; w2; b2; r$, and all idle loops id_{0_i} in M_c to the idle loop id_{0_1} in M_a; loops $\mathsf{id}_{\$_1}$ and $b1; w2$ in M_c to $\mathsf{id}_{\$_1}$ in M_a; and all *work*- and *buy*-instances in M_c are mapped to similar instances in M_a. The label part of the morphism m maps labels *rest* and id_0 in M_c to id_0 in M_a, and *work* to *work*, *buy* to *buy*, and $\mathsf{id}_\$$ to $\mathsf{id}_\$$. Note that both components of m map idle loops to idle loops, i.e., are functors.

Model M_a is indeed an abstraction of M_c as several non-idle transitions between states 0_1, 0_2, 0_3 in M_c are mapped to the same element — identity id_{0_1} in M_a. That is, morphism m forces a rich sub-LTS of M_c, which consists of states $0_{1,2,3}$ and transitions between them, to collapse into a primitive sub-LTS of M_a consisting of a single state 0_1 with its idle loop. It means that the notion of being idle is relative: idling in one model can be a complex activity in another model.

Thus, idle loops and mappings using them are useful for behavior modeling, and we can finally define our notion of an LTS.

Definition 4 (LTSs). An *LTS* is a triple $M = (T, \lambda, L)$ with T and L categories of transitions and labels resp., and λ: $T \to L$ a functor.

An *LTS morphism* m: $M \to M'$ is a pair of functors commuting with labeling as shown in diagram Fig. 8. □

4.3 $\mathcal{M}$-sets and their colimits

We considered several structures (graphs, categories, LTSs), consisting of sets and functions between them such that certain equations hold. Morphisms between these structures are defined *componentwise*, i.e., as families of functions between the constituent sets. As function composition is associative, all morphisms we considered are also associatively composable, which, along with identity morphisms (identity functions for all components), give us categories Graph, Cat, LTS of graphs, categories, LTSs, resp. This sequence should begin with the simplest category of this kind: the category Set of all sets and functions.

In this section, we define $\mathcal{M}$-sets—a generalization of structures mentioned above, and their colimits. $\mathcal{M}$-sets are a remarkable construct. On the one hand, they are a far reaching generalization of structures we considered above (graphs, pre- and categories, LTSs), which encompasses any model defined by an MOF-metamodel, whose constraints can be expressed by equations. On the other hand, $\mathcal{M}$-sets can be seen as an immediate generalization of sets, such that operations on $\mathcal{M}$-sets are very close to operations over sets; we will employ this for defining $\mathcal{M}$-sets colimits.

4.3.1 $\mathcal{M}$-sets. Let $\mathcal{M}$=$(G_\mathcal{M}, E_\mathcal{M})$ be a metamodel consisting of a directed graph $G_\mathcal{M}$ and a set $E_\mathcal{M}$ of equational constraints (equations). An *instance* of $\mathcal{M}$ is a pair $I = (G_I, t_I)$ with G_I a graph specifying the instance's data, and

t_I: $G_I \to G_{\mathcal{M}}$ a graph morphism called *typing mapping* such that all equations in $E_{\mathcal{M}}$ hold (and we write $I \models E_{\mathcal{M}}$).

For an element $e \in G_I$, its *type* is an element $t_I(e) \in G_{\mathcal{M}}$. Conversely, for an element (type) $x \in G_{\mathcal{M}}$, its *extension* is a set $[\![x]\!]^I$ of all those elements of G_I, whose type is x. If x is a node n in $G_{\mathcal{M}}$, then $[\![n]\!]^I$ is a set of nodes in G_I whose type is n. If x is an arrow a: $n \to n'$ in $G_{\mathcal{M}}$, then $[\![a]\!]^I \subset [\![n]\!]^I \times [\![n']\!]^I$ is a relation defined by arrows in G_I whose type is a; indeed, each such arrow gives us a pair from $[\![n]\!]^I \times [\![n']\!]^I$ (to simplify presentation, we ignore the possibility of multi-relations, when the same pair of nodes in G_I is related by more than arrow of the same type). Also, we assume that all arrows a in $G_{\mathcal{M}}$ have multiplicities that force relations $[\![a]\!]^I$ to be total single-valued functions; otherwise, we replace a "bad" arrow by a span whose legs are functions (this is analogous to the known procedure for the relational schema normalization). In this way, we build a mapping $[\![\_]\!]^I$: $G_{\mathcal{M}} \to \mathsf{Set}$, whose image in Set actually gives us the instance graph. It is easy to check that mapping $[\![\_]\!]^I$ is a graph morphism.[3]

Thus, instances of $\mathcal{M}$ are systems of sets and functions similar to those mentioned above, and we call them $\mathcal{M}$*-sets*. For example, an LTS is an $\mathcal{M}$-set for the metamodel $\mathcal{M}_{\mathsf{LTS}}$ specified in Fig. 10, where two equality symbols denote two commutativity constraints expressing functoriality of λ. (The graph of this metamodel is also shown in the upper box in Fig. 8.)

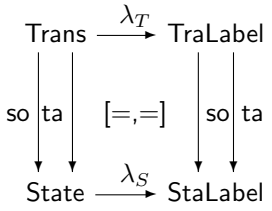


Fig. 10. LTS metamodel

Morphism of $\mathcal{M}$-sets are defined componentwise, i.e., a morphism f: $I \to J$ is a family of functions f_n: $[\![n]\!]^I \to [\![n]\!]^J$ indexed by nodes n in $G_{\mathcal{M}}$, such that $[\![a]\!]^I ; f_n = f_m ; [\![a]\!]^J$ for all arrows a: $m \to n$ in $G_{\mathcal{M}}$. An equivalent non-indexed definition is a graph mapping f: $G_I \to G_J$ commuting with typing, $f ; t_J = t_I$. Thus, we have a category $\mathcal{M}$-Set of $\mathcal{M}$-sets and their morphisms (a categorician would call it the *presheaf topos* of shape $\mathcal{M}$ [2]). For example, each of the categories Graph, Cat, LTS can be seen as category $\mathcal{M}$-Set for the respective metamodel $\mathcal{M}$.

A fundamental fact about $\mathcal{M}$-sets is that they can be perfectly managed in parallel with ordinary sets. Ordinary notions of union, product, factorization (quotient set), etc, are easily defined for $\mathcal{M}$-sets componentwise. Below we will see how this can be employed for defining colimit.

4.3.2 Colimits of $\mathcal{M}$-sets. Figure 11 describes the general case of merging two LTSs as described in Sect. 3. Nodes in the diagram are LTSs, and arrows are their morphisms as specified by Fig. 8. The binary colimit operation takes a span (r_1, R, r_2) of LTSs as its input, and produces a *covering* cospan (u_1, U, u_2) (note

[3] Moreover, we can also freely extend graph $G_{\mathcal{M}}$ with all possible arrow compositions and identity loops, and thus make it a category $G^+_{\mathcal{M}}$; then mapping $[\![\_]\!]^I$ can be freely extended to a functor $[\![\_]\!]^I$: $G^+_{\mathcal{M}} \to \mathsf{Set}$.

the label [cover]), which makes the entire square commutative (label [=]) (these and other properties were discussed in Sect. 3. Now we will give a constructive definition of colimit and show that it provides all the necessary properties.

We first consider the simplest case of set colimit, when nodes in the diagram are interpreted by sets, and arrows by functions. To compute colimit, we take the disjoint union of sets, $M = M_1 \uplus M_2$, and factorize it by the equivalence relation $E^* \subset M \times M$, which is the reflexive transitive closure of relation $E = \{(r_1(x), r_2(x)) \colon x \in R\}$. That is, we define $U \stackrel{\text{def}}{=} M/E^*$ and $u_i \stackrel{\text{def}}{=} \iota_i; \varepsilon$ $(i = 1, 2)$, where $\iota_i \colon M_i \to M$ are canonical injections, and $\varepsilon \colon M \to M/E^*$ is the canonical surjection. As all these are defined up to isomorphism, U is also so defined. It is easy to see that $r_1; u_1 = r_2; u_2$ by definition of E^* and u_i, and that (u_1, u_2) is a cover since ε is a cover.

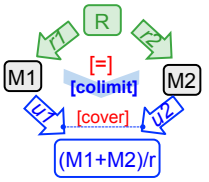


Fig. 11. Colimit pattern

A remarkable property of U is its *minimality*: for any cospan $M_1 \xrightarrow{v_1} V \xleftarrow{v_2} M_2$ such that $r_1; v_1 = r_2; v_2$, there is a *unique* function $! \colon U \to V$ such that $u_i; ! = v_i$ for $i = 1, 2$. It can be proved that minimality is equivalent to coverage: a cospan U is covering iff it is minimal in the sense above. Indeed, if set U would contain an element e beyond the union of images of u_i, mapping $!$ could map this e to any element of V without destroying the commutativity conditions. It can be also proved by standard categorical arguments that minimality in the sense above uniquely determines the colimit cospan up to a unique isomoprhism between the heads. Thus, colimit can be declared declaratively as the uniquely defined minimal cospan completing the input span up to a commutative square as shown in Fig. 11.

Now we define colimit of $\mathcal{M}$-sets, i.e., for the same diagram Fig. 11 but interpreted by $\mathcal{M}$-sets for some given metamodel $\mathcal{M} = (G_{\mathcal{M}}, E_{\mathcal{M}})$. Everything will be done componentwise. As $\mathcal{M}$-set morphisms are families of functions indexed by nodes in $G_{\mathcal{M}}$, each such node n determines a Set-interpretation of the span R, i.e., as a span $M_1^n \xleftarrow{r_1^n} R^n \xrightarrow{r_2^n} M_2^n$ of two functions r_i^n between the respective sets. (For example, for LTSs, $G_{\mathcal{M}}$ has four nodes, and for each of them we have a span, whose legs are defined in Fig. 8.) Hence, as we have already defined colimit for sets, for any n we have a colimit set cospan $M_1^n \xrightarrow{u_1^n} U^n \xleftarrow{u_2^n} M_2^n$. A tedious "equation chasing" shows that owing to (i) commutativity constraints for $\mathcal{M}$-set morphisms, and (ii) minimality of colimit, for any arrow $a \colon n \to n'$ in the metamodel, there is a unique function $a^! \colon U^n \to U^{n'}$ commuting with everything necessary, so that actually we have a cospan of $\mathcal{M}$-sets and their morphisms: $M_1 \xrightarrow{u_1} U \xleftarrow{u_2} M_2$.

For example for the LTS metamodel in Fig. 10, for $a = \mathsf{so} \colon \mathsf{Trans} \to \mathsf{State}$, we first compute cospan

$$M_1^{\mathsf{Trans}} \xrightarrow{\mathsf{so}_1; u_1^{\mathsf{State}}} U^{\mathsf{State}} \xleftarrow{\mathsf{so}_2; u_2^{\mathsf{State}}} M_2^{\mathsf{Trans}}$$

such that $r_1^{\mathsf{Trans}}; \mathsf{so}_1; u_1^{\mathsf{State}} = r_2^{\mathsf{Trans}}; \mathsf{so}_2; u_2^{\mathsf{State}}$, and then use minimality of U^{Trans} to compute $\mathsf{so}^!\colon U^{\mathsf{Trans}} \to U^{\mathsf{State}}$. Arrows $\mathsf{ta}^!\colon U^{\mathsf{Trans}} \to U^{\mathsf{State}}$, $\mathsf{so}^!\colon U^{\mathsf{TraLabel}} \to U^{\mathsf{StaLabel}}$, and $\mathsf{ta}^!\colon U^{\mathsf{TraLabel}} \to U^{\mathsf{StaLabel}}$ are computed in the same way.

The same idea also works for computing arrows $\lambda_S^!\colon U^{\mathsf{State}} \to U^{\mathsf{StaLabel}}$ and $\lambda_T^!\colon U^{\mathsf{Trans}} \to U^{\mathsf{TraLabel}}$. For example, for $\lambda_S^!$, we first compute the cospan

$$M_1^{\mathsf{State}} \xrightarrow{\lambda_{S1}; u_1^{\mathsf{StaLabel}}} U^{\mathsf{StaLabel}} \xleftarrow{\lambda_{S2}; u_2^{\mathsf{StaLabel}}} M_2^{\mathsf{State}},$$

and then use minimality of U^{State} to compute $\lambda_S^!\colon U^{\mathsf{State}} \to U^{\mathsf{StaLabel}}$. In this way we build an instance $U\colon G_{\mathcal{M}} \to \mathsf{Set}$ of the graph $G_{\mathcal{M}}$ specified in Fig. 10, which satisfy the necessary commutativity conditions and, hence, is a valid instance of the metamodel $\mathcal{M}_{\mathsf{LTS}}$.

Mappings $u_i\colon M_i \to U$, $(i = 1, 2)$ between $\mathcal{M}$-sets are defined in a straightforward way. Moreover, yet another round of equation chasing shows that cospan (u_1, U, u_2) is actually the minimal one amongst all cospans that complete the span (r_1, R, r_2) up to a commutative square in the category of $\mathcal{M}$-sets. Hence, by a standard categorical reasoning, U is defined up to an isomorphism. Finally, in Appendix, we describe an algorithm for computing colimit of an arbitrary multiary span specifying a match between multiple $\mathcal{M}$-sets (like, e.g., in Fig. 3).

Thus, however complex are (a) the metamodel $\mathcal{M} = (G_{\mathcal{M}}, E_{\mathcal{M}})$ and (b) the input configuration (match) of $\mathcal{M}$-sets, colimit of the latter is a simple operation computed componentwise. Note how well the diagram Fig. 11 works: it is abstract enough and hides all "n-layers" and hence is applicable for any category of $\mathcal{M}$-sets, but it is concrete enough to show how each layer works.

4.3.3 Three colors of model merge. The analysis above makes it clear that model merge consists of two parts. The first is model match, which requires heuristics, analysis of names and modeling contexts, and is not a fully automatic, hence, not an algebraic, operation. After models are matched, their merge is a routine automatic procedure called colimit. In a sense, the latter is trivial, hence, the slogan: *CT makes trivial things really trivial*, and, thus, facilitates the fundamental for software engineering *separation of concerns*. In our diagrams, we distinguish between pseudo-algebraic and algebraic (automatic) operations by coloring their results in green and blue, resp. We believe that this green-blue separation of concerns is important for model merge and other MMt operations specified in [12].

5 Parallel composition and Limits

Parallel composition of executable components (or behaviors) is a fundamental operation of behavior modeling. It assumes that several (local) components run simultaneously so that a global state is a tuple of local states, and a global transition is a tuple of local transitions. Some local transitions may be required to be synchronized, that is, be always executed simultaneously (like a *handshake*). We will show that parallel composition can be defined categorically by an operation

called *limit*, which is dual to colimit in some precise sense (so that synchronization appears as a *parallel match*, and parallel composition as a *parallel merge*).

We will begin with a simple case of unsynchronized composition, then consider how synchronization is specified by mappings, and finally discuss duality between the merge and parallel composition.

5.1 Unsynchronized composition

Suppose that Ben and Bill from Sect. 3.1 decided to give their behavior models a try, and run them simultaneously, both starting at the same moment, but otherwise entirely independently. It means, for example, that if one of them works, the other can do nothing (idle), or also work, or eat if he has already earned his dollar. Then the parallel composition consists of all possible pairs of states, and all possible pairs of transitions as shown in Fig. 12. Importantly, in order to specify the possibility of, say, Ben is working while Bill is idling, we added to every state in Bill's model an idle (loop) transition, and the same for Ben's model, which makes them categories. Now we can describe the parallel merge U as a Cartesian product of two categories: horizontal and vertical transitions are those in which one component is idling, whereas diagonal transitions combine two non-idle ones. (And the idle loops in the product are pairs of idle loops.) By the definition of a product category, all triangle diagrams are commutative, which makes perfect sense in our context. For example, commutativity of the two left top triangles says that the differences between the three ways of earning two dollars by the system U are irrelevant from the global viewpoint. There are two canonical projection mappings $u_i\colon U \to M_i$, which are (total and single-valued as we assume by default, and) surjective. Surjectivity of projections captures simultaneity of the parallel merge: any local element (state or transition) must become a component of a global element. Another important postcondition is that any global element is actually a pair of local elements, that is, if for global elements $e, e' \in U$, equalities $u_1(e) = u_1(e')$ and $u_2(e) = u_2(e')$ hold, then $e = e'$. We denote this property by a predicate label [key]: in the relational algebra jargon, the pair of attributes u_1, u_2 is a key for the "binary relation" U.

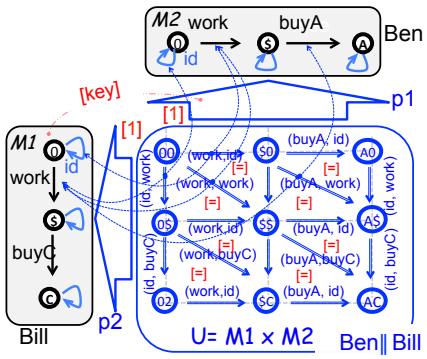


Fig. 12. Parallel composition (product)

Finally, the parallel merge U is, of course, defined up to a canonical isomorphism: OIDs of elements in U can be arbitrarily chosen. Hence, we claim that defining local LTSs to be categories, and their parallel merge to be their product, is an appropriate mathematical model of the scenario.

5.2 Synchronization via Cospans

Now suppose that to earn their dollars, Ben and Bill need to work together on a joint project. After that, they act separately and indepedently, as shown by LTS U in Fig. 13. Remarkably, this LTS can be automatically computed by an operation called *limit*, if synchronization between the models is properly specified. This specification is given by mappings $s_i\colon M_i \to S$, $(i = 1, 2)$ from the local models to a common model S representing the global (synchronized) view of the behavior. The global view should evidently contain a global transition *work*, composed from two local instances of *work* acting in parallel (note the corresponding links in mappings s_i). In addition, there are two global idle loops. The first one, id_0, is a pair of local idles, $(\mathsf{id}_0@M_1, \mathsf{id}_0@M_2)$ as shown by the respective links in the cospan (s_1, s_2). The second global identity, id_2, is much more interesting. It is the image of several local transitions from both sides, which means that from the global viewpoint, the differences between the respective states and actions are not essential (compare with the case in Fig. 9).

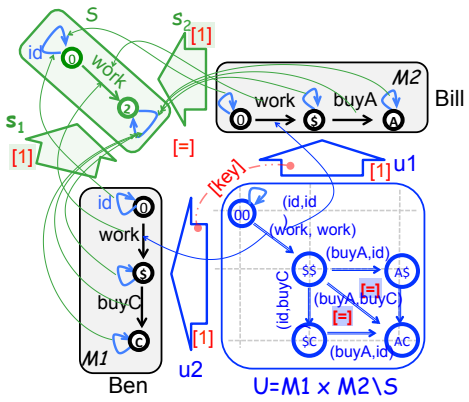


Fig. 13. Synchronized parallel composition

Now the desired result of synchronized composition can be formally defined by a simple formula: $U \stackrel{\text{def}}{=} \{e \in M_1 \times M_2\colon\ s_1(u_1(e)) = s_2(u_2(e))\}$, where u_1, u_2 are canonic projections described above. The operation producing the product span (U, u_1, u_2) from a cospan (S, s_1, s_2) of LTSs and their morphisms as described above is called *limit* (in the category of LTSs), that is, span (u_1, U, u_2) is the limit of cospan (s_1, S, s_2) as shown by diagram Fig. 14.

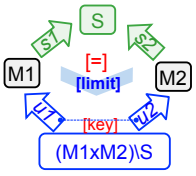


Fig. 14. Limit pattern

Note a remarkable duality between colimit and limit operations given by diagrams in Fig. 11 and Fig. 14: the diagrams are mutually convertible by inverting directions of all morphisms. Match is given by a span while synchronization is given by a cospan. Merge is the result of stepwise factorization (colimit) $A \to A/E_1^* \to (A/E_1^*)/E_2^* \to \ldots \to ((A/E_1^*)/\ldots) = A/E^*$, while parallel composition is the result of stepwise subtraction (limit) $M_1 \times M_2 \supset U_1 \supset U_2 \supset \ldots \supset U_n = U$. The results of these dual processes are also remarkably dual: the colimit cospan enjoys the joint cover property and minimality, while the limit span enjoys the joint key property and maximality (it can be shown that properties cover and key are dual in some precise sense). A precise formal account of limit-colimit duality can be found in any CT textbook.

The limit-colimit duality has a practical consequence. Our green-blue separation of concerns for merge immediately carries on to parallel composition: synchronizing models by cospans is non-trivial (green), whereas composition as such is automatic and trivial (blue). In general, the limit-colimit duality gives rise to a sort of technology transfer: some merge technologies should be adaptable for parallel composition and conversely.

6 Structural Modeling

The goal of this section is to show that categorical constructs considered above for LTSs also make sense for structural modeling, e.g., we argue that merging class diagrams (CDs) motivates considering sequential composition of associations, and identity associations. In other words, it makes sense to consider a CD D as a (finite) representation of (perhaps, infinite) category D^+ generated by D by composing its associations.

Figure 15 shows a simple CD with three classes and three given directed associations between them, 'worksFor', 'locatedAt', and 'commutesTo'. Sequential composition of the two upper association (note the label [;]) produces a derived association shown with dashed (blue) arrow. Comparison of this derived association with given association 'commutesTo' is an important property of the domain described by the CD. If both associations are equal, the addresses a person commutesTo are exactly the addresses of the person's work places. This constraints can be expressed by placing equality symbol [=] between the two arrows. If the derived association subsets 'commutesTo' (note the middle dotted arrow labeled [≤]), a person can commute to addresses besides person's work places. If the derived association supersets 'commutesTo' (the rightmost dotted arrow), some persons can reach their work places without commuting (e.g., with carpooling). The latter two constraints are known in UML as subsetting of associations, and are usually shown with arrows between the association edges as we did in Fig. **??**. Note a difference between the UML and the CT mind-sets. The constraints we considered above are known in UML, and can be precisely specified in OCL; however, the UML does not anyhow encourage to think about association composition and the corresponding constraints like discussed above. In contrast, the categorical view on CDs does encourage the modeler to think about the composition of associations and possible commutativity or subsetting constraints.

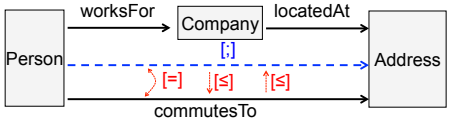


Fig. 15. Composition of associations and three possible constraints

Another use of association composition is for managing semantic relativism, when as association basic in one CD is a derived association in another CD (similarly to what we discussed above for transitions).

Figure 16 presents an example of merging two class diagrams, CD1 and CD2. Correspondence links state that classes Person and Employee refer to the same class of real world objects, and association 'commutes' in CD2 corresponds to the

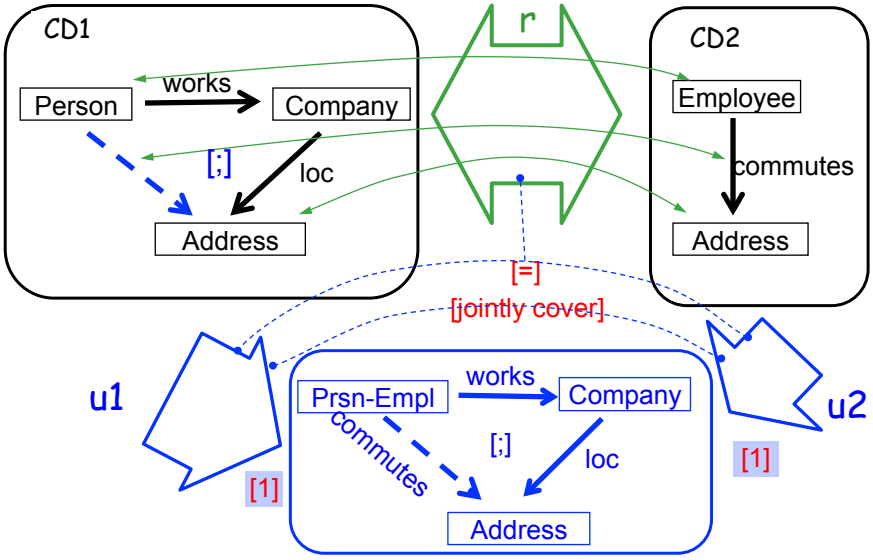


Fig. 16. Map-based merge of class diagrams

sequential composition of associations 'works' and 'loc' in CD1. Hence, model CD2 is, in fact, a view to model CD1, and hence their merge is isomorphic to CD1. Note that without the link from 'commutes' in CD2 to the derived association in CD1, association 'commutes' would appear as a basic element in the merge model thus creating redundancy.

7 Discussion and Conclusions

We will briefly summarize our work in the paper to broaden applicability of our observations. We will also point out to several important problems left for the future work, and conclude the paper.

7.1 Mappings, their management, and arrow encapsulation. The CT analysis of MMt is based on inter-model mappings. An MMt scenario amounts to a sequence of operations over models and mappings, and important properties of the scenario can be expressed via properties of mappings. (Note that focusing on mappings is not foreign for MMt: it just makes explicit traceability mappings, which are often considered auxiliary and left implicit in typical non-categorical specifications used in MMt.) Complex scenarios give rise to complex systems of mappings, but CT provides an effective mechanism to manage this complexity—*arrow encapsulation (AE)*.

The essence of AE is that complex configurations of mappings are encoded by arrows, which can be composed and from which new configurations can be built, and so on. On the other hand, many models themselves can be decomposed into more elementary blocks and their mappings, and so on until the truly primitive level of elements and links is reached. AE thus provides zoom-in and zoom-out mechanism based on algebra, which allows the user to switch between abstraction

levels as needed. Figure 8, along with Figures 11 and 14, present a very simple example; other examples can be found in [12].

AE also allows the user to relate properties of arrow configurations on different levels, thus supporting validity checks of a scenario implementation against its specification. Moreover, CT provides a library of well-designed patterns for AE (see [12]), and techniques that allow us to reason about them.

7.2 Unification. Categorical specification of MMt scenarios provides three facets of unification. First, categorical operations may be defined across different types of input configurations, e.g., the same colimit operation provides a mathematical model of merging two models with an inter-model span, or several models with a system of inter-model spans, including multi-ary spans and spans between spans (see [14]). Colimit works equally well for injective or surjective single-valued mappings, and even for multi-valued mappings, if the latter are replaced by spans whose legs are single-valued (this is analogous to the well-known normalization of relational schemas). Similar arguments work for parallel composition via limits (not surprising as we have seen that limits are dual to colimits).

Second, categorical operations can be defined generically w.r.t. different metamodels. Moreover, for any metamodel $\mathcal{M}$ being a graph with equational constraints, colimits and limits of $\mathcal{M}$'s instances are computed in basically the same way irrespective of the shape of the graph. On the other hand, managing more complex constraints needs special "tuning" and can be challenging. However, categorical logic provides several patterns that allow us to replace complex constraints with quantifiers by equations (via introduction of new objects and mappings). We plan to address this issue in the future work.

Third, a large diversity of MMt scenarios can be composed from a small library of elementary categorical operations: arrow composition, limits and colimits. Adding to this library a higher-order operation of building powerobjects makes it even more expressible.

7.3 Usability. CT provides a powerful formal framework than binds MMt together, and inevitably brings with it a vast array of formal constructs and challenging usability issues. This is a large area of future research, and we can only make several comments. The first one is quite general. As said in the introduction, we see the role of CT for MMt as being similar to the role of the relational data model for data management, and with methodologically similar usability issues. The database community successfully solved these issues and replaced the mathematical relational theory by its engineering approximations, e.g., SQL, relational algebra as it is taught in standard database courses, and ER-modeling. We believe that a similar path can well be followed by the MDE community as soon as it starts detecting the categorical patterns hidden inside the MMt scenarios (see [12] for a discussion). More concretely, CT formalities come with a well-designed modularization mechanism (that we tried to demonstrate), which can greatly facilitate practical use of the CT-framework. Finally, when the formal semantics is well-understood, we can approach creating a reasonable engineering notation in a principled way.

For instance, we can introduce various notational macros. A simple example is shown in the inset figure to be compared with Fig. 9. In the former, idle loops are omitted (they can always be restored if needed), and a marker [id] inside a chain of consecutive transitions says that the composition of transitions is equal to the idle loop of the state closest to the marker. Two such markers inside a chain formed by two opposite transitions (as in model M_a) require the transitions to be mutually inverse, and can be replaced by a single marker [inv]. More examples of how a categorical semantics can simplify a notation can be found in [10].

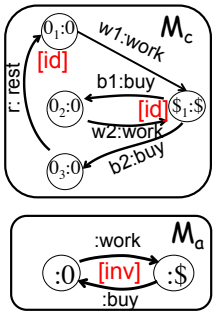


Fig. 17. Syntactic tips

Instead of conclusion. Playing with LTSs in this paper was conceived purely for its tutorial role. Unexpectedly, these toy examples revealed several technical details of LTSs and operations over them that appear to be novel. This can be seen as evidence of CT's effectiveness as a modeling language: even a mere rearrangement of a known area in categorical terms can lead to new insights into the subject. CT is actually a way of thinking about multi-structural problems like those encountered in MMt, which enhances understanding of the domain and suggest reasonable architectural patterns for designing MMt tools.

References

1. Anjorin, A., Schürr, A., Taentzer, G.: Construction of integrity preserving triple graph grammars. In: Ehrig et al. [15], pp. 356–370
2. Barr, M., Wells, C.: Category Theory for Computing Science. Prentice Hall International Series in Computer Science (1995)
3. Batory, D.S., Azanza, M., Saraiva, J.: The objects and arrows of computational design. In: MoDELS. LNCS, vol. 5301, pp. 1–20. Springer (2008)
4. Batory, D.S., Latimer, E., Azanza, M.: Teaching model driven engineering from a relational database perspective. In: Moreira et al. [23], pp. 121–137
5. Bernstein, P.A.: Applying model management to classical meta data problems. In: CIDR (2003)
6. Boronat, A., Knapp, A., Meseguer, J., Wirsing, M.: What is a multi-modeling language? In: WADT. LNCS, vol. 5486, pp. 71–87. Springer (2009)
7. Diskin, Z.: Model synchronization: mappings, tile algebra, and categories. In: R. Lämmel et al. (ed.) Postproceedings GTTSE 2009. LNCS#6491, Springer (2011)
8. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, queries, and kleisli categories. In: de Lara, J., Zisman, A. (eds.) FASE. LNCS, vol. 7212, pp. 163–177. Springer (2012)
9. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. JOT 10, 6: 1–25 (2011)
10. Diskin, Z.: Visualization vs. specification in diagrammatic notations: A case study with the uml. In: Hegarty, M., Meyer, B., Narayanan, N.H. (eds.) Diagrams. LNCS, vol. 2317, pp. 112–115. Springer (2002)
11. Diskin, Z.: Mathematics of generic specifications for model management. In: Rivero, L.C., Doorn, J.H., Ferraggine, V.E. (eds.) Encyclopedia of Database Technologies and Applications, pp. 351–366. Idea Group (2005)
12. Diskin, Z., Kokaly, S., Maibaum, T.: Mapping-aware megamodeling: Design patterns and laws. In: Erwig, M., Paige, R.F., Wyk, E.V. (eds.) SLE. Lecture Notes in Computer Science, vol. 8225, pp. 322–343. Springer (2013)
13. Diskin, Z., Maibaum, T., Czarnecki, K.: Towards category theory foundations for model management. Tech. Rep. GSDLab-TR 2014-03-03, University of Waterloo/McMaster University, http://gsd.uwaterloo.ca/node/566 (2014)
14. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying overlaps of heterogeneous models for global consistency checking. In: MoDELS Workshops. LNCS, vol. 6627, pp. 165–179. Springer (2010)
15. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G. (eds.): Graph Transformations - 6th International Conference, ICGT 2012, Bremen, Germany, September 24-29, 2012. Proceedings, Lecture Notes in Computer Science, vol. 7562. Springer (2012)
16. Fiadeiro, J.L., Costa, J.F., Sernadas, A., Maibaum, T.S.E.: Process semantics of temporal logic specifications. In: Bidoit, M., Choppy, C. (eds.) COMPASS/ADT. Lecture Notes in Computer Science, vol. 655, pp. 236–253. Springer (1991)
17. Goguen, J.A.: A categorical manifesto. Mathematical Structures in Computer Science 1(1), 49–67 (1991)
18. Golas, U., Lambers, L., Ehrig, H., Giese, H.: Toward bridging the gap between formal foundations and current practice for triple graph grammars - flexible relations between source and target elements. In: Ehrig et al. [15], pp. 141–155
19. Große-Rhode, M.: Semantic Integration of Heterogeneous Software Specifications. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2004)

20. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of model synchronization based on triple graph grammars. In: Whittle, J., Clark, T., Kühne, T. (eds.) MoDELS. LNCS, vol. 6981, pp. 668–682. Springer (2011)
21. Lambers, L., Hildebrandt, S., Giese, H., Orejas, F.: Attribute handling for bidirectional model transformations: The triple graph grammar case. ECEASST 49 (2012)
22. Lauder, M., Anjorin, A., Varró, G., Schürr, A.: Bidirectional model transformation with precedence triple graph grammars. In: Vallecillo, A., Tolvanen, J.P., Kindler, E., Störrle, H., Kolovos, D.S. (eds.) ECMFA. Lecture Notes in Computer Science, vol. 7349, pp. 287–302. Springer (2012)
23. Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P.J. (eds.): Model-Driven Engineering Languages and Systems - 16th International Conference, MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings, Lecture Notes in Computer Science, vol. 8107. Springer (2013)
24. Rossini, A., de Lara, J., Guerra, E., Rutle, A., Lamo, Y.: A graph transformation-based semantics for deep metamodelling. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE. LNCS, vol. 7233, pp. 19–34. Springer (2011)
25. Rossini, A., Rutle, A., Lamo, Y., Wolter, U.: A formalisation of the copy-modify-merge approach to version control in mde. J. Log. Algebr. Program. 79(7), 636–658 (2010)
26. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: A diagrammatic formalisation of mof-based modelling languages. In: TOOLS. LNBIP, vol. 33. Springer (2009)
27. Sabetzadeh, M., Easterbrook, S.: An algebraic framework for merging incomplete and inconsistent views. In: 13th Int.Conference on Requirement Engineering (2005)
28. Taentzer, G., Mantz, F., Arendt, T., Lamo, Y.: Customizable model migration schemes for meta-model evolutions with multiplicity changes. In: Moreira et al. [23], pp. 254–270

A Appendix. Multi-ary Colimits of $\mathcal{M}$-Sets

In this section, we consider a general case of merging multiple models related by multiple spans. First of all, we note that a set of local models and intermodel spans amounts to a bigger set of *secondary* models and morphisms. For example, for the case in Fig. 2 two local models with a span amount to three secondary models M_i, R_{ij} and two morphisms; for Fig. 3, we have eight secondary models and eight secondary moprhisms. The operation called *colimit* can be considered as a procedure that first builds a disjoint union of all secondary models, and then factorizes it by gluing together elements linked together by secondary morphisms.

Given sets $(A_1, .., A_m)$ and functions $(f_1, .., f_k)$ between them, **begin:**
1) let $A = \uplus_{i=1...m} A_i$ and
   ι_i: $A_i \to A$ are canonic injections
2) let $E = \varnothing \subset A \times A$
3) for every f_j, $j = 1, \ldots, k$ **do**
   for every a in the domain of f_j **do**
   let $E = E \cup \{(a, f_j(a))\}$ **od od**
4) let E^* = reflexive transitive closure of E
5) let ϵ: $A \to A/E^*$ be the canonic surjection
6) **return** multicospan with head $U = A/E^*$
   and legs $u_i = \iota_i; \epsilon$: $A_i \to U$, $i = 1 \ldots m$

Fig. 18. Colimit algorithm for multiple sets and functions

A precise definition of the algorithm for the case when models are sets, and morphisms are (total single-valued) functions, is specified in Fig. 18. Two required post-conditions (commutativity and joint coverage) are easy to check: $f_j; u_{i'} = u_i$ for any f_j: $A_i \to A_{i'}$, and $\bigcup_{i=1..m} u_i(A_i) = U$. Cospan $(U, u_1 ... u_m)$ has two important properties. First, the result of factorization, and hence, the head U, are defined up to a canonic isomorphism: if $U' = A/E^*$ and u'_i: $A_i \to U'$ is another cospan returned by the procedure, them there is a uniquely defined bijection b: $U \to U'$ such that $u_i; b = u'_i$ for all $i = 1..m$. The second property is minimality: for any cospan $(V, v_i$: $A_i \to V)$ with head V commuting with all functions f_j as described above (i.e., $f_j; v_{i'} = v_i$), there is a unique function $!$: $U \to V$ such that $u_i; ! = v_i$ for all $i = 1...m$. It can be proved that minimality is equivalent to coverage: a cospan U is covering iff it is minimal in the sense above.