1

Coproducts in Categorical Information System Specification

Michael Johnson[1] and Robert Rosebrugh[2]

[1] Macquarie University, Sydney, Australia,
mike@ics.mq.edu.au
[2] Mount Allison University, NB, Canada
rrosebru@mta.ca

Abstract. The specification of information systems using categorical techniques can be based on sketches that use finite limits and finite coproducts. The use of finite limits is a well-understood extension of classical universal algebra, but the use of coproducts takes the theory outside normal algebraic specification techniques. In particular, coproduct specifications: may not have initial models; are not preserved by Kan extensions; and are not supported by standard update procedures. Nevertheless, coproducts have proved very important in industrial applications of the specification techniques. This paper argues the case for their inclusion in information system specifications, and treats each of the three difficulties enumerated above.
Keywords: Specification, category theory, information system

1 Introduction

Categorical Information System Specification (CISS) is a methodology which is based on categorical universal algebra and algebraic specification techniques, and which has been used with considerable success in industrial consultancies (including for example [7], [8]). In fact it was initially developed to support a very large information system project [6].

The CISS method specifies an information system by giving a presentation (a sketch) of a category with finite limits and finite coproducts. Colomb, and independently Paré, have asked why coproducts are needed and how CISS deals with some of the following issues.

Specifications which include coproducts:

1. May not have initial models
2. Have models which need not be preserved by Kan extensions
3. Cannot be modified by standard update procedures.

This paper answers those questions.

* Research partially supported by the Australian Research Council, NSERC Canada, and the Oxford Computing Laboratory.

In Section 2 we outline the mathematical foundations of CISS defining in particular EA-sketches which are the main specification tool. That section also discusses the models of EA-sketches, including the existence of initial models. Section 3 develops an example to give some indication of the constraint specification techniques used. In fact much of categorical logic can be expressed in an EA-sketch, and so a very wide range of constraints can be incorporated into CISS specifications, but for this paper we comment on the importance, both in real world modelling terms and in technical terms, of coproducts in the specification. In Section 4 we introduce a new theory of database updates based on Kan extensions. Although Kan extensions of models of EA-sketched specifications need not be models, because Kan extensions need not preserve non-connected limits and colimits, an updated model will be a certain sort of epimorphic image of the Kan extension of the original model. Finally, Section 5 provides update procedures for models of EA-sketches with coproduct specifications. We conclude that, while the utility of CISS has already been demonstrated in industrial consultancies, the *theory* of CISS should support, and can support, coproduct specifications.

We are grateful to our colleague, C.N.G Dampney, for inspiring much of our work on information system specification, and to Bob Colomb and Bob Paré for asking the questions that this paper answers.

2 Categorical information system specification

This section describes the foundation of our technique for category-theoretic information system specification (CISS). It is motivated by standard category-theoretic algebraic specification techniques. We assume some familiarity with elementary category theory, as might be obtained in [2], [17] or [24]. The graphs we use will always be what have sometimes been called "directed multi-graphs, possibly with loops" (see for example [2] page 7). The limits and colimits we will deal with will all be finite in our applications.

Definition 1. *A* cone $C = (C_b, C_v)$ *in a graph* G *consists of a graph* I *and a graph morphism* $C_b : I \longrightarrow G$ *(the* base *of* C*), a node* C_v *of* G *(the* vertex *of* C*) and, for each node* i *in* I*, an edge* $e_i : C_v \longrightarrow C_b i$*.* Cocones *are defined dually. The edges* e_i *in a cone (respectively cocone) are called* projections *(respectively* injections*).*

Definition 2. *A* sketch $\mathbb{E} = (G, \mathbf{D}, \mathcal{L}, \mathcal{C})$ *consists of a graph* G*, a set* $\mathbf{D}$ *of pairs of directed paths in* G *with common source and target (called the commutative diagrams) and sets of cones (*$\mathcal{L}$*) and cocones (*$\mathcal{C}$*) in* G*.*

Every category has an underlying sketch: Let G be the underlying graph of the category, $\mathbf{D}$ the set of all commutative diagrams, and $\mathcal{L}$ (respectively $\mathcal{C}$) the set of all limit cones (respectively colimit cocones). Of course underlying sketches are usually not small. The advantage of the theory of sketches is that we can frequently use a sketch to give a finite presentation of an infinite category.

Definition 3. *Let* $\mathbb{E} = (G, \mathbf{D}, \mathcal{L}, \mathcal{C})$ *and* $\mathbb{E}' = (G', \mathbf{D}', \mathcal{L}', \mathcal{C}')$ *be sketches. A* sketch morphism $h : \mathbb{E} \longrightarrow \mathbb{E}'$ *is a graph morphism* $G \longrightarrow G'$ *which carries, by composition, diagrams in* $\mathbf{D}$*, cones in* $\mathcal{L}$ *and cocones in* $\mathcal{C}$ *to respectively diagrams in* $\mathbf{D}'$*, cones in* $\mathcal{L}'$ *and cocones in* $\mathcal{C}'$*.*

Definition 4. *A* model M *of a sketch* $\mathbb{E}$ *in a category* $\mathbf{S}$ *is a graph morphism from* G *to the underlying graph of the category* $\mathbf{S}$ *such that the images of pairs of paths in* $\mathbf{D}$ *have equal composites in* $\mathbf{S}$ *and cones (respectively cocones) in* $\mathcal{L}$ *(respectively in* $\mathcal{C}$*) have images which are limit cones (respectively colimit cocones) in* $\mathbf{S}$*.*

Equivalently, a model is a sketch morphism from $\mathbb{E}$ to the underlying sketch of the category $\mathbf{S}$. (Generally, $\mathbf{S}$ will be taken to be *lextensive* [3] because lextensive categories provide the basis for the kind of categorical logic we use in industrial specifications.) We can also express models in terms of functors as follows.

Write $\widetilde{\mathbb{E}}$ for the category generated by the graph G of $\mathbb{E}$, subject to the relations $\mathbf{D}$ of $\mathbb{E}$. Using the evident inclusion $G \longrightarrow \widetilde{\mathbb{E}}$ we will sometimes refer to nodes of G as objects, edges of G as arrows and (co)cones of $\mathbb{E}$ as (co)cones in $\widetilde{\mathbb{E}}$.

If M is a model of $\mathbb{E}$ in $\mathbf{S}$ then M extends to a functor $\widetilde{M} : \widetilde{\mathbb{E}} \longrightarrow \mathbf{S}$.

Definition 5. *If* M *and* M' *are models a* homomorphism $\phi : M \longrightarrow M'$ *is a natural transformation from* $\widetilde{M}$ *to* $\widetilde{M}'$*. Models and homomorphisms determine a category of models of* $\mathbb{E}$ *in* $\mathbf{S}$ *denoted by* $\mathrm{Mod}(\mathbb{E}, \mathbf{S})$*, a full subcategory of the functor category* $[\widetilde{\mathbb{E}}, \mathbf{S}]$*.*

Formal specification techniques which could be based on sketches have been used for a variety of purposes ([13], [14], [21], [12] provide a number of examples) but they have particularly been used for the specification of abstract datatypes. In that case, the existence of canonical models is usually very important. In contrast, the models of sketches that we use for information system specification will generally *not* be canonical and should not be. A model will merely be the state of the information system at a particular point in time.

To emphasise the distinction between datatype specification and information system specification: A datatype specification might be used to present, as an initial model, a datatype of say lists; in the same context an information system specification would be presenting, as a model, a list. Of course information systems are highly structured data objects, and single instances consist of many sets and many functions between them, so the analogy oversimplifies the nature of information system specification. Nevertheless, it clarifies the distinction.

In fact, real world models usually have no constants except for attribute values (attributes are defined formally below). In that situation, initial models do exist, and they correspond to 'empty' information systems — those in which all sets, except attribute sets, are empty. In other words, initial models are information systems in which no information is (or has yet been) stored.

It turns out that we do not need the full generality of sketches to specify information systems so we will now restrict to the notion of EA-sketch (short for 'entity-attribute' sketch).

We speak of (limit-class, colimit-class) sketches when $\mathcal{L}$ and $\mathcal{C}$ are required to contain (co)cones only from the specified (co)limit-classes. For example, (finite-product, $\emptyset$) sketches correspond to (multi-sorted) algebraic theories.

Definition 6. *An* EA sketch $\mathbb{E} = (G, \mathbf{D}, \mathcal{L}, \mathcal{C})$ *is a (finite limit, finite coproduct) sketch such that*

- *There is a specified cone with empty base in* $\mathcal{L}$*. Its vertex will be called* 1*. The object* 1 *is not the codomain of an arrow. Arrows with domain* 1 *are called* elements.
- *Nodes which are vertices of cocones whose injections are elements are called* attributes*. An attribute is not the domain of an arrow. Nodes which are neither attributes, nor* 1*, are called* entities*.*
- *The graph of* G *is finite and acyclic.*

Definition 7. *A* database state D *for an EA sketch* $\mathbb{E}$ *is a model of* $\mathbb{E}$ *in* $\mathbf{Set}_0$*, the category of finite sets. The* category of database states of $\mathbb{E}$ *is the category of models* $\mathrm{Mod}(\mathbb{E}, \mathbf{Set}_0)$ *of* $\mathbb{E}$ *in* $\mathbf{Set}_0$*. Thus morphisms of database states are natural transformations. For brevity we sometimes denote the category of database states by* $\mathrm{Mod}(\mathbb{E})$*.*

EA-sketches can be viewed as an extension of entity-relationship (ER) models [4], [23]. An ER diagram can be used to generate in a systematic way a graph for an EA-sketch (the details are dealt with in [16]). The theory [2] corresponding to an EA-sketch includes objects representing the queries (first noted in [9]). The extra "semantic" power of CISS comes from the non-graph components: $\mathbf{D}$ can be used to record constraints, and $\mathcal{L}$ and $\mathcal{C}$ can be used to calculate query results from other objects, and these query results can in turn be used to add constraints, etc. It is not surprising that this extra power, by providing a much richer set of possibilities for specifying constraints, is of benefit in information modelling, and it has been the basis of the successful consultancies which have used CISS as a formal information system specification technique.

3 CISS and coproducts

We develop an example of an EA sketch and discuss the importance of coproducts.

Example 1. Figure 1 is part of the graph of an artificial EA sketch, derived from a fragment of a real sketch data model [7]. The other components of this EA sketch are as follows: $\mathbf{D}$ contains both triangles; $\mathcal{L}$ contains the empty cone with vertex 1 (not shown), the square (whose diagonal, one of the projections, is also not shown) and three further cones intended to ensure that the three arrows indicated $\rightarrowtail$ are realised as monics; and $\mathcal{C}$ contains the cocone with vertex

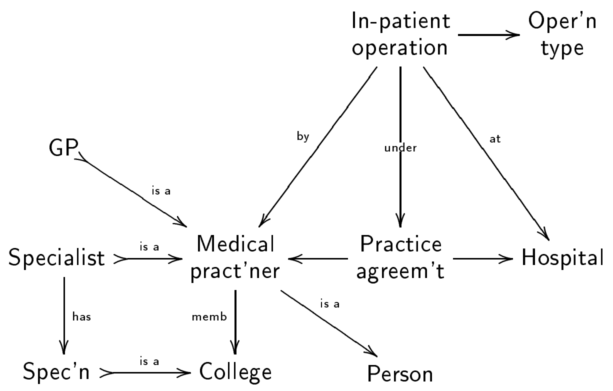

Fig. 1. A fragment of a graph for an EA sketch

Medical practitioner and base Specialist and GP (short for General practitioner), along with a number of cocones with attributes (not shown) as vertices.

Briefly, we expand on each of the components in turn, indicating what they, as a specification, correspond to in models.

The graph is a type diagram. The three monic arrows indicate subtypes. The other arrows are functions (methods) which given an instance of their domain type will return an instance of their codomain type. The names on nodes and arrows have no formal significance but do indicate the real world semantics being captured in the specification and will be used from now on in our discussion.

The commutativity of the two triangles represents a typical real-world constraint: Every in-patient operation conducted at a particular hospital by a particular medical practitioner must take place under a practice agreement (a type of contract) between that hospital and that practitioner. If, instead the left hand triangle were not required to commute (that is, was not in $\mathbf{D}$) then it would still be the case that every operation took place under an agreement, but Dr X could operate under Dr Y's practice agreement.

The inclusion of the square in $\mathcal{L}$ ensures that in models it will be a pullback. This ensures that the specialists are precisely those medical practitioners who are members of a college which occurs in the subtype Specialisation. This is important because the registration procedures (not shown) for specialists are different from those for other medical practitioners. Similar pullbacks can be used to specify other subtypes, for example, the medical practitioners with a specific specialisation, say paediatricians.

Subtype inclusion arrows, and other arrows that are required to be monic in models, are so specified using pullbacks. Incidentally, we could just include two such cones since elementary properties of pullbacks ensure that if the square is realised as a pullback and its bottom arrow as a monic then its top arrow will necessarily be monic. Notice also that the arrow into Person is not required to be monic. A single person might appear more than once as a medical practitioner, as for example, when the person practises both as a GP and as a specialist, or practises in more than one specialisation. This is a minor point, but the distinction between people and the roles they play is an important distinction in many real world applications.

The cocone with vertex Medical practitioner ensures that the collection of medical practitioners is the disjoint union of the collection of specialists and the collection of GPs. Specifications which don't include this constraint could be used to allow the storage of details about medical practitioners for whom specialisation or GP status is unknown or inapplicable.

As is common practice, attributes are not shown in Figure 1, but they are important. They are usually large fixed value sets, often of type `integer` (with specified bounding values), `string` (of specified maximum length), `date` etc. Some examples for this model include the validity period of a practice agreement, the name and the address of a person, the classification of a hospital, the date of an operation, the provider number of a medical practitioner and many more. Strictly, they are all part of the graph, but in practice they are usually listed separately in a data dictionary.

A database state or model of the EA sketch of Example 1 is a collection of finite sets and functions satisfying the constraints. The set corresponding to, for example, In-patient operation should be thought of as the collection of all operations currently stored in the information system.

We now note the importance of coproducts in an information system specification using an EA-sketch. There are two main points: Coproducts are technically essential, since the absence of other constants in most models makes it impossible to access the data without using attributes, which being coproducts of 1, do have constants as elements; and coproducts are practically desirable because (especially following the growth of object oriented modelling and its integration with data modelling via UML [20]) systems are very commonly modelled using type hierarchies and these often involve (on analysis) coproduct types.

4 Update theory and Kan extensions

How can we tell mathematically whether one database state is an elementary update (corresponding to a single instance insertion or deletion) of another database state? This section provides a precise answer.

As is common practice, we will not deal theoretically with updates which modify existing values. Instead, such modifications are expected to be obtained by (elementary) deletions and insertions. Furthermore, a formalism for deletions can be obtained from our treatment of insertions. Thus, we focus our attention on elementary insertions.

Next, a few words about *cascades*. There remains some controversy in the database community about whether elementary updates should cascade. (In the following it is convenient to speak about entities and relations even though relations are themselves entities in our framework — whenever $X \longrightarrow Y$ we can say that X is a relation involving the entity Y.) Suppose we attempt to delete an instance of an entity, which is related to another instance of another entity. Some systems will prohibit this operation, requiring the relation instance to be deleted first, while other systems will cascade the delete, deleting the relation instance (and any relation instances that it takes part in etc.) automatically. Similarly insertions may cascade (an insertion of a relation instance may automatically lead to the insertion of a new entity instance to be related by the relation instance, etc.) or the system may prohibit cascades and require that all instances to be related are inserted before the relation instance can be inserted.

Now to the formal treatment of insertions. We will insert a new instance x in a database state D at a given entity E.

Let $\mathbb{E}$ be an EA sketch, and E be an object of $\mathbb{E}$. We define $\mathbb{E}[x_E]$ to be $\mathbb{E}$ augmented with a new arrow $x : 1 \longrightarrow E$ (and no new commutative diagrams, cones or cocones). Write J for the evident inclusion $\widetilde{\mathbb{E}} \longrightarrow \widetilde{\mathbb{E}[x_E]}$.

Write $\mathrm{Mod}(\mathbb{E})$ for the category of database states. Thus objects of $\mathrm{Mod}(\mathbb{E})$ are certain functors $D : \widetilde{\mathbb{E}} \longrightarrow \mathbf{Set}_0$. Write, abusing notation, $U = J^*$ for the functor $[\widetilde{\mathbb{E}[x_E]}, \mathbf{Set}_0] \longrightarrow [\widetilde{\mathbb{E}}, \mathbf{Set}_0]$ given by composition with J, and for its restriction $U : \mathrm{Mod}(\mathbb{E}[x_E]) \longrightarrow \mathrm{Mod}(\mathbb{E})$. Now J^* has a left adjoint computed as the left Kan extension of the functor along the inclusion $J : \widetilde{\mathbb{E}} \longrightarrow \widetilde{\mathbb{E}[x_E]}$. Call the left adjoint L and let η be the unit of the adjunction.

Because the Kan extension of a model need not be a model, in particular it may not take discrete (co)cones to (co)products, we need to take some care dealing with natural transformations $\phi : LD \longrightarrow D'$ even when D and D' are database states (of $\mathbb{E}$ and $\mathbb{E}[x_E]$ respectively). We call such a ϕ a *base epi* when its components ϕ_F are epis for all objects F of $\mathbb{E}$ that are not the vertex of a cone, nor of a cocone.

Definition 8. *Suppose D is an $\mathbb{E}$ database state, E an object of $\mathbb{E}$, and D' an $\mathbb{E}[x_E]$ database state. Then $m : D \rightarrowtail UD'$ is an* insert update *of D at E if there exists a base epi $\phi : LD \longrightarrow D'$ such that*

$$\begin{tikzcd} D \arrow[rr, "\eta_D"] \arrow[dr, tail, "m"'] & & ULD \arrow[dl, "U\phi"] \\ & UD' & \end{tikzcd}$$

commutes.

Remark 1. i) Intuitively, LD should be thought of as the universal database state containing D and a new constant x of type E. The definition requires D' to be "squeezed" between D and LD, coherently with the way D is included in both D' and LD.

ii) Our definition permits, but does not mandate, cascades. The Kan extension, LD, cascades maximally. Requiring D' to be a base epimorphic image of LD means that what in the Kan extension is a cascaded insert, may in D' be a

relation instance assignment to an already extant entity instance or attribute value.
iii) In fact, LD will not in general be a database state since the sketch may include coproduct cocones. This is easy to see by supposing that there is an attribute A_E (a fixed coproduct of 1s) for E and by using the Kan extension formula [17] (page 236) to calculate the value of $LD(A_E)$. As shown by Lemma 1 the "maximal cascade" mentioned above corresponds to $LD(A_E) = D(A_E) + 1$, a different coproduct of 1s, and so LD can't be a database state. Nevertheless, D' is a database state, and ϕ indicates how cascaded inserts which have disrupted coproducts in LD are replaced by assignments to extant instances or values in D'.
iv) Elementary properties of adjunctions ensure that if ϕ, as in the definition, exists, then it is unique.

Lemma 1. *For E' not equal to E*

$$LD(E') = D(E') + \{\beta : E \longrightarrow E'\}$$

Proof. $LD(E') = \mathrm{colim}(J/E' \longrightarrow \widetilde{\mathbb{E}} \longrightarrow \mathbf{Set}_0)$ where the first arrow is the projection from the comma category, and the second is the database state D. Since J is bijective on objects, each object E' of $\widetilde{\mathbb{E}}[x_E]$ is of the form JF for some F. Consider J/JF, whose objects are of the form $(A, \alpha : JA \longrightarrow JF)$. Such α are either Ja for $a : A \longrightarrow F$, or $\beta x : J1 \longrightarrow JF$ with $\beta = Jb$, some b. Notice that there are no morphisms in J/JF to or from objects of the second kind (because there are no commutative diagrams in $\mathbb{E}[x_E]$ involving x) and that $D(1) = 1$. Furthermore, the identity on JF is terminal among the other objects in J/JF. Thus the colimit is the sum as claimed.

5 Coproduct updates

We provide pseudocode procedures for insert and delete updates of models of EA sketches with coproducts. For the purposes of this paper we seek procedures which can deal with coproducts, so for simplicity we assume that the EA sketch has no limit cones except the limit of the empty cone (required by Definition 6). The procedures illustrate the inadequacy of ordinary update algorithms for dealing with coproduct specifications, as well as showing the feasibility of updating such specifications using new procedures.

The insert procedure deals with a database state `D`, and constructs a new database state `D'`. It thus requires storage for sets, which appear in the procedure as, for example, `D'(E)`, and for functions, which are presumed to be stored as their graphs (ordered pairs of (domain, codomain) values). As an example of a function, the line `D'(x_E)1 = t` says that the pair $(1, t)$ should be added to the graph of the function corresponding to x_E in the database (partial) state `D'`.

The procedure begins by adding (if necessary) the element `t` and then defines, or if necessary prompts the user to specify, all the images of the new `t`.

```
/* INSERT PROCEDURE FOR EA SKETCHES WITH COCONES

Algorithm INS(t, D, E, ins, D')
/*  pre: D a database state, E an entity of D, and t not in D(E).
/* post: if ins is FALSE, D is unchanged, and D' is garbage;
/*       if ins is TRUE t is in D'(E), and D' is a database state for
/*       the extended sketch E[x].

D' = D                    /* D' will be the new database state
D'(E) = D'(E) + { t }     /* This is union, not disjoint union.
D'(x_E)1 = t              /* Abuse notation writing 1 for the element of 1
ins = TRUE
for each a : E --> E'
    /* Loop completion means all D'(a)t are defined
    if D'(a)t is defined
         next a
    elseif a is an injection for a cocone C
       and there exists t' in D'(E') not in the image of the base of C
       /* (For say M+M = N+N+N this avoids blowup)
       /* There is an element of the coproduct available for D'(a)t.
         for each such t'   /* Try to assign D'(a)t=t'
             EXTEND_FUNCTION(D', a, t, t', ins)
             if ins = TRUE  /* We have succeeded so we're done for this a
                 break
         if ins = FALSE     /* We have failed, because of CD constraints
             new(t' : E')   /* so we need an entirely new element.
             D'(a)t = t'
             INS(t', D', E', ins, D') /* (Assts to D'(x_E')1 will be ignored)
    elseif D'(E') empty
         /* hence need to augment D'(E') with new element
         new(t': E')
         D'(a)t = t'
         INS(t', D', E', ins, D')     /* (Assts to D'(x_E')1 will be ignored)
    else
         INTER_ASSIGN(D', a, t, ins)
         /* Let user choose D'(a)t's value. Does not allow new element creation
    if not ins
         exit   /* On fail we still have previous state D and we exit.

for each cocone C with vertex E
    if there exists j: E'--> E in C and t' in D'(E') with D'(j)t' = t
        next C
    elseif there exists j : E' --> E in C and t' in D'(E') with D'(j)t' = undef
       /* (for M+M = N+N+N this avoids blowup)
       /* There is a base element available to send to t.
        for each such j
            EXTEND_FUNCTION(D', j, t', t, ins)
            if ins = TRUE
                break
        if ins = FALSE   /* No appropriate t' available.
            output('Choose an injection j in cocone C')
            input(j : E' --> E) /* User selects injection j in the cocone
            new(t' : E')        /* and we generate a new element
            D'(j)t' = t         /* corresponding via j to t.
            INS(t', D', E', ins, D') /* (Assts to D'(x_E')1 will be ignored)
    else
        output('Choose an injection j in cocone C')
        input(j : E' --> E) /* user selects injection to the cocone
        new(t':E')
        D'(j)t' = t
        INS(t', D', E', ins, D') /* (Assts to D'(x_E')1 will be ignored)
    if not ins
        exit   /* On fail we must give up.
```

Fig. 2. An insert update algorithm for EA sketches with cocones (and no cones except for the empty cone)

The proposed images are checked to ensure that commutative diagrams are preserved (and this needs to be done carefully, because at this point many of the functions are only partial functions). The user is also given an abort option. This is necessary since it is possible to begin an insert which can never be completed without violating commutative diagram constraints.

The most delicate aspect of the procedure is the avoidance of infinite recursion when inserting into an entity which is the vertex of two distinct cocones, especially if the bases include repeated elements. When for example E is the vertex of a cocone with base M and M, and also of a cone with base N, N and N, the procedure will in fact insert six new elements into E.

Routine, if tedious, verifications, along with Lemma 1 allow us to prove the following propositions which link the update theory and the procedures.

Proposition 1. *If the procedure* `INS` *returns with* `ins = TRUE`*, then* `D'` *is a database state for* $\mathbb{E}[x_E]$.

Proposition 2. *If the algorithm* `INS` *returns* `TRUE`*, and if no object is the vertex of more than one cocone, then there exists a base epi* $\phi : LD \longrightarrow D'$ *in* $\mathrm{Mod}(\mathbb{E}[x_e])$ *such that in* $\mathrm{Mod}(\mathbb{E})$*, with* m *the evident inclusion,*

$$m : D \rightarrowtail UD' \longleftarrow ULD : U\phi$$

and $(U\phi)\eta_D = m$.

6 Conclusion

The use of category theory in the specification and analysis of information systems is growing (a selection of other recent work in the area includes [15], [1], [18], [19], [10], [11]) and the theoretical work is being applied in industry ([8], [7], [22]). But the use of cocones in EA-sketch specifications, and of coproducts in their models, raises a number of questions which we have treated here.

Coproduct specifications:

1. May not have initial models, but we have argued that this is much less important than in other specification frameworks, and that in fact EA sketches do usually have initial models;
2. Are not preserved by Kan extensions, and we do use Kan extensions in our new theoretical treatment of updates. Nevertheless, for EA sketches the empty limit is preserved by Kan extensions (by Lemma 1 and Definition 6 which prohibited arrows into its vertex), and any attributes' and coproduct entities' extra instances will be corrected by the transformation ϕ (Definition 8);
3. Are not supported by standard update procedures, so we have provided procedures designed to update EA-sketched models with coproduct specifications (Figure 2, and in the Appendix Figures 3 and 4).

We have also pointed out the need for coproduct specifications to provide attributes with known elements for querying, and to support coproduct based models which are becoming more widely used in industry.

Ongoing theoretical work aims to link more closely the update theory of Section 4 and the procedures in Section 5, and to extend the procedures to sketches including non-empty limit specifications and prove that they satisfy formally specified update properties based on the update theory. Meanwhile, practical work is continuing in projects on computational category theory which will add to the CASE tools currently being used for CISS development in industry.

References

1. K. Baklawski, D. Dimovici and W. White. A categorical approach to database semantics. *Mathematical Structures in Computer Science*, 4:147–183, 1994.
2. M. Barr and C. Wells. *Category theory for computing science*. Prentice-Hall, second edition, 1995.
3. Aurelio Carboni, Steven Lack, and R. F. C. Walters. Introduction to extensive and distributive categories. *Journal of Pure and Applied Algebra*, 84:145–158, 1993.
4. P. P. -S. Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 2:9–36, 1976.
5. E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13:377–387, 1970.
6. C. N. G. Dampney and Michael Johnson. TIME Compliant Corporate Data Model Validation. Unpublished report to Telecom Australia, 1991.
7. C. N. G. Dampney and Michael Johnson. Fibrations and the DoH Data Model. Unpublished report to NSW Department of Health, 1999.
8. C. N. G. Dampney, Michael Johnson and G. M. McGrath. Audit and Enhancement of the Caltex Information Strategy Planning (CISP) Project. Unpublished report to Caltex, 1993.
9. C. N. G. Dampney, Michael Johnson, and G. P. Monro. An illustrated mathematical foundation for ERA. In *The unified computation laboratory*, pages 77–84, Oxford University Press, 1992.
10. Zinovy Diskin and Boris Cadish. Algebraic graph-based approach to management of multidatabase systems. In *Proceedings of The Second International Workshop on Next Generation Information Technologies and Systems (NGITS '95)*, 1995.
11. Zinovy Diskin and Boris Cadish. Variable set semantics for generalised sketches: Why ER is more object oriented than OO. In *Data and Knowledge Engineering*, to appear, 2000.
12. H. Ehrig and B. Mahr. *Fundamentals of algebraic specifications*. Springer-Verlag, 1985.
13. J. Goguen, J. W. Thatcher and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. Yeh, *Current Trends in Programming Methodology IV*, 80–149, Prentice-Hall, 1978.
14. J. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24:68–95, 1977.
15. A. Islam and W. Phoa. Categorical models of relational databases I: Fibrational formulation, schema integration. Proceedings of the TACS94. Eds M. Hagiya and J. C. Mitchell. *Lecture Notes in Computer Science*, 789:618–641, 1994.

16. Michael Johnson and C. N. G. Dampney. On the value of commutative diagrams in information modelling. In *Springer Workshops in Computing*, Springer-Verlag, 1994.
17. Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5, Springer-Verlag, 1971.
18. Frank Piessens and Eric Steegmans. Categorical data specifications. *Theory and Applications of Categories*, 1:156–173, 1995.
19. F. Piessens and Eric Steegmans. Selective Attribute Elimination for Categorical Data Specifications. Proceedings of the 6th International AMAST. Ed. Michael Johnson. *Lecture Notes in Computer Science*, 1349:424-436, 1997.
20. J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
21. H. Reichel. *Initial Computability, Algebraic Specifications and Partial Algebras.* Clarendon Press, 1987.
22. G. Southon, C. Sauer, and C. N. G. Dampney. Lessons from a failed information systems initiative: issues for complex organisations *International Journal of Medical Informatics*, Elsevier Science, 1999.
23. J. D. Ullman. *Principles of Database and Knowledge-Base Systems.* Volume 1, Computer Science Press, 1988.
24. R. F. C. Walters. *Categories and Computer Science.* Cambridge University Press, 1991.

Appendix: Deletes and utility procedures

For completeness we append here a delete procedure, and the extension of the theory of inserts to deal with deletes, along with the utility procedures needed for the insert update procedure.

```
/* DELETE PROCEDURE FOR EA SKETCHES WITH COCONES

Algorithm DEL(t, D, E)
/* The cases used below are to ensure termination.

if t not in D(E)
   exit
case type(E) of
  cocone vertex:
       for a : E' --> E with a not injection, for t' with D(a)t' = t
          DEL(t', D, E')
       for injections i, t' such that i(t') = t  /* There may be more than one.
          REMOVE(t, D, E)   /* This order to avoid possible infinite recursion.
          DEL(t', D, E')
  other:
       for each i : E --> E' injection
          DEL(i(t), E', D)
       for each a : E' --> E, for each t' with D(a)t' = t
          DEL(t', D, E')
       REMOVE(t, D, E)
/* end of DEL

Algorithm REMOVE(t, D, E)
/* Removes t and all definitions D(a)t = t'.

for a : E --> E'
   D(a)t = undefined    /* Means (t,t') removed from the graph of D(a).
D(E) = D(E) - { t }
/* end of REMOVE
```

Fig. 3. A delete update algorithm for EA sketches with cocones (and no cones except for the empty cone)

The delete procedure for coproduct EA sketches (Figure 4) is a recursive cascading delete. This is necessary since deleting from either a cocone vertex or a cocone base entity requires deletion from the other, and this in turn may recur to arbitrary (finite) depth. The order of `REMOVE` and `DEL` is important.

Although we don't need it for this paper, for completeness we outline now how to get a formal theory of delete updates in a manner dual to the formal theory of insert updates. We aim to say that $D \rightarrowtail D'$ is a delete from D' when it is an insert into D. However, because cascades happen in opposite directions, x as used above points to the "back" of a delete cascade, so the correspondence between inserts and deletes is a little delicate. In detail: $D \rightarrowtail D'$ is a delete update from D' at E' when there exists an E such that $D \rightarrowtail D'$ is an insert update of D at E (and E need not be E'). In fact, an elementary delete could correspond to a multiple insert (if for example two different inserts at E and E'' cascade to one common final instance at E') and vice versa. So, $D \rightarrowtail D'$ is also a delete update from D' at E' when there exists a family $(E_i)_{i\in I}$ so that $D \rightarrowtail D'$ can be obtained by iterated inserts at E_i in D. Of course, in the absence of cascades the correspondence is straightforward.

```
/* UTILITY PROCEDURES FOR INSERT UPDATE ALGORITHM

Algorithm CHCD(D', a, t, t') : Boolean
/* returns TRUE if the definition D'(a)t = t' respects commutative diagrams
/* we check separately CD's with both paths beginning a, since D'(a)t not yet
/* defined (such paths would be checked twice but one if is false)

CHCD = TRUE
for each CD = (pa, p'a)     /* p, p' are paths; both beginning with a
  if not (D'(p)t' = undef) and not (D'(p')t' = undef) and not ( D'(p)t' = D'(p')t' )
      /* execute if we can test for comm and the test fails
      CHCD = FALSE
for each CD = (pa, p') or (p', pa)   /* p, p' are paths
  if not (D'(p)t' = undef) and not (D'(p')t = undef) and not ( D'(p)t' = D'(p')t )
      /* execute if we can test for comm and the test fails
      CHCD = FALSE
/* end of CHCD

Algorithm EXTEND_FUNCTION(D', a, t, t', asn)
/* Checks cd's and extends D'(a) by t |--> t' if ok

if CHCD(D', a, t, t')
   D'(a)t = t'
   asn = TRUE
else
   Output('CD violation, extend function failed')
   asn = FALSE
/* end of EXTEND_FUNCTION

Algorithm INTER_ASSIGN(D', a, t, asn)
/* Interactively asks user for a valid value of D'(a)t.

asn = FALSE
repeat
   Output('type value of D'(a)t in D'(E') or CTRL-Q to quit')
   Input(t')  /* t' assumed to be of type E' or CTRL-Q.
   if t' = CTRL-Q
      exit    /* Needed since it's possible to attempt an insert
              /* which would never satisfy CD requirements.
   EXTEND_FUNCTION(D', a, t, t', asn)
until asn = TRUE                    /* So user must have succeeded.
/* end of INTER_ASSIGN
```

Fig. 4. Utility procedures for the insert update algorithm