ENTERPRISE INFORMATION SYSTEMS: SPECIFYING THE LINKS AMONG PROJECT DATA MODELS USING CATEGORY THEORY

Michael Johnson
Department of Computing, Macquarie University, Sydney, Australia
Email: mike@ics.mq.edu.au

C.N.G. Dampney
Department of Computing, Macquarie University, Sydney, Australia
Email: kit@ics.mq.edu.au

Keywords: Enterprise information system, logical data independence, category theory, view update, interoperating systems, data modelling

Abstract: Major enterprise information systems are frequently specified by integrating information models which have been developed in separate divisions of the enterprise. These models, often called project data models, embody the information that needs to be modelled in the enterprise, but they can be difficult to link because when common information is stored in different divisions it is frequently stored in significantly different forms. This paper describes a new technique, based on category theory, that uses the specification of logically data independent views to link project data models. The link mechanism is powerful because the use of views permits data in radically different forms to be linked, and a new solution to the view update problem allows the linking mechanism to be embodied as code, thus allowing the linked data models to be implemented as interoperating information systems. The paper is somewhat theoretical since it describes the foundation for a new technique, but the methods described here are being tested in large consultancies including the enterprise models for a government department, an oil company, and a telecommunications carrier.

1. INTRODUCTION

Major enterprise information systems are frequently specified by integrating information models which have been developed in separate divisions of the enterprise. These models, often called project data models, embody the information that needs to be modelled in the enterprise, but they are often difficult to link because when common information is stored in different divisions it is frequently stored in significantly different forms.

This paper describes a technique that uses the specification of logically data independent views to link project data models. The link mechanism is powerful because the use of views permits data in radically different forms to be linked, and a new solution to the view update problem [10], [4], allows the linking mechanism to be embodied as code, thus allowing the linked data models to be implemented as interoperating information systems.

The tools that we use are based on category theory. Category theory ([2], [11], [15]) is a branch of mathematics renowned for its semantic power, its simple axiom set, and its use of graphical techniques. It has been widely used for specification in computer science in, for example, abstract data types [7], [8], semantics of programming languages [12], [13], and functional programming [1].

We make extensive use of logical data independence [14]. Logical data independence supports views of data with different logical structure from the original model. We argue that this is especially important in specifying links between project data models as it is only rarely that such models include common data in the *same* logical structure. We present below a mathematical treatment of logically independent views in a category theoretic framework and show how it is used to support the linking of project data models to create an enterprise data model.

In summary our technique is as follows. Given two project data models which we wish to link we use category theoretic specification techniques to develop a single logically data independent model of (parts of) each of the project data models. This model, $\mathbb{K}$, is the link. It is a single formal representation of those parts of the original systems that share common data in whatever form. It remains to develop the interoperation code.

To develop the interoperation code we need to explore the *transmissibility* of update information from each project data model, "out" to the common submodel $\mathbb{K}$ and back "in" to the other project data model. It turns out that the outward transmissibility (from a project model to $\mathbb{K}$) is guaranteed for formal mathematical reasons. Inward transmissibility on the other hand *cannot* be guaranteed. Instead, inward transmissibility needs to be explored in detail using view updatability, and a novel approach to view updatability is outlined below.

The plan of the paper is as follows. In Section 2 we set the scene with an example illustrating the category theoretic information system specification techniques that we use. In Section 3 we define views in our category theoretic framework. Section 4 explains how for these views the view update problem can be analysed. Finally in Section 5 we show how the theory developed in the earlier sections can be used to link interacting project data models. and in Section 6 we illustrate this using our example specification introduced in Section 2.

2. CATEGORY THEORETIC SPECIFICATION: AN EXAMPLE

In this section we give an example to illustrate some of the category theoretic tools we use in our information systems consultancy work, and we introduce some basic terminology. The main problem that we will consider here is the *specification* of information systems, including particularly the specification of real-world constraints. The example that we have chosen is artificial. It has been constructed so as to illustrate the main constraint specification tools with a minimum of extraneous details. Nevertheless, despite being small it is sufficiently realistic that we will return to it in section 6 to illustrate project data model linking.

We will not review elementary category theoretic notions. Definitions and basic properties of commuting diagrams, limits, and coproducts, can be found in most introductions to category theory, including [2] and [15]. We will assume familiarity with ER terminology [14].

The graph shown in Figure 1 is a part of a specification for an e-commerce information system. The graph may be thought of as an ER model: The vertices of the graph represent entities (types), the arrows represent many-to-one mandatory relations (functions or methods), and the arrow with tail ($\rightarrowtail$) represents a sub-entity (subtype or injective function). There is a correspondence between standard ER models and such graphs given by tabulating many-to-many relations and using subentities to eliminate non-mandatory many-to-one relations. For simplicity attributes have not been shown in Figure 1, with the sole exception of Price, but if desired they can be included as arrows from the entity on which they are defined to a node representing their attribute value set, or they can be recorded in a separate data dictionary.

Of course, the specification technique we use also has extra-ER aspects. We specify three further sets of properties: Which diagrams commute, which nodes occur as limits of specified diagrams (via specified cones) defined in the graph, and which nodes occur as coproducts (via specified cocones) of specified nodes of the graph. In our example the extra-ER properties include the specification that the two triangles commute, that the square be a pullback (a kind of limit), and that the Product node be the coproduct of Physical product and Virtual product (electronically available software, music, etc). These constraints ensure that: Deliveries only take place to fill specific Orders (the commuting triangles); All Products, and only those Products, which have a Physical location are Physical products (the pullback); All Products are either Physical products or Virtual products (the coproduct). The Location entity stores both warehouse locations and URLs.

In our technique the quadruple consisting of the graph, sometimes called the *underlying graph*, and the extra-ER constraints (three sets), is called a schema. Formally

Definition 1 A *schema* $\mathbb{E} = (G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ is a directed graph G, a set $\mathcal{D}$ of pairs of paths in G with common source and target, a set $\mathcal{L}$ of cones in G, and a set $\mathcal{C}$ of cocones in G.

(It may be worth noting that the definition uses merely directed graphs — there is no need to decorate some arrows with tails to indicate subentities.

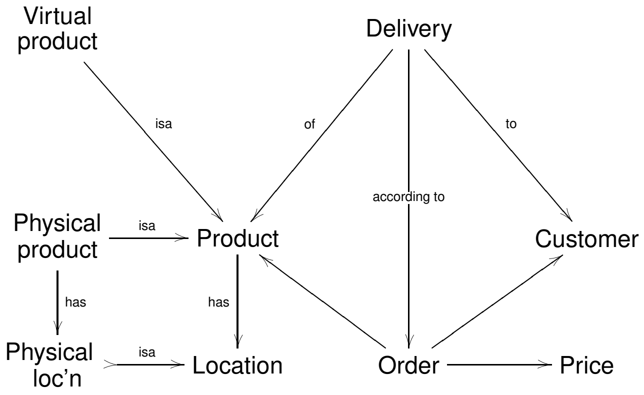

Figure 1. An illustrative example: Part of an e-commerce schema

This follows from the simple technical result that the property of being a subentity can also be specified as a limit. See for example [10] page 539.)

A schema can be thought of as the raw material, the base entities and relationships, for a mathematical abstraction called the *classifying category*. Roughly speaking it is the smallest category containing the schema, satisfying the constraints, and closed under finite limits and finite coproducts. The classifying category has objects corresponding to each of the structural queries that can be applied to the information system [3]. The classifying category for $\mathbb{E}$ is denoted by $Q(\mathbb{E})$ to remind us that it incorporates all the queries. The category $Q(\mathbb{E})$ is usually infinite, so it is not likely to be very useful in implementations, but because it embodies all the structural queries it has theoretical value as it represents all possible data that can be derived from an information system, and because it is finitely generated it remains tractable.

Schemata can be interrelated using schema maps. A *schema map* is a graph morphism between the corresponding graphs which maps each of the constraints on the first schema to a constraint (already) specified in the second schema. Formally

Definition 2 Consider two schemata $\mathbb{E} = (G, \mathcal{D}, \mathcal{L}, \mathcal{C})$ and $\mathbb{E}' = (G', \mathcal{D}', \mathcal{L}', \mathcal{C}')$. A *schema map* $m : \mathbb{E} \longrightarrow \mathbb{E}'$ is a graph morphism, also denoted $m : G \longrightarrow G'$ which carries by composition diagrams in $\mathcal{D}$, cones in $\mathcal{L}$ and cocones in $\mathcal{C}$ to respectively diagrams in $\mathcal{D}'$, cones in $\mathcal{L}'$ and cocones in $\mathcal{C}'$.

Sometimes we will speak of a schema map from a schema into a category. In that case we mean a schema map from the schema into the underlying schema of the category: its graph is the underlying graph of the category, and its constraints are all of the constraints that happen to be true in the category — all of the commuting diagrams, all of the limits, and all of the coproducts.

An information system, sometimes called a database state or instance, is an assignment of, for every node in the schema a finite set (the set of instances or values of that entity or attribute), and for every arrow in the schema a function between the corresponding sets (the relationships among the entity instances or the attribute values corresponding to the instances), such that the constraints are satisfied in the category of finite sets. Often we will elide the reference to database and simply refer to a database state as a state. Formally

Definition 3 A *state* E for a schema $\mathbb{E}$ is a schema map $\mathbb{E} \longrightarrow \mathbf{Set}_0$ into the category of finite sets.

In other words, a state for a given schema is a diagram of (finite) sets and functions which is the same shape as the schema's underlying graph, and which satisfies the constraints.

Now that we have established the context in which we work, we turn to the important question of defining views of schemata.

3. DEFINABLE VIEWS

The question of which views are definable for a particular schema is long-standing, and has generated considerable controversy. The main problem has been that proposals for definable views have frequently been formulated to correspond to those view mechanisms which have been implemented in a particular system.

Instead, we seek to deal with views defined as widely as possible.

One of us has argued [9] that truly logically independent views should have the following properties:

1. A logically independent view should be able to contain any data that can be derived from data stored anywhere in the entire information system. This is sometimes referred to as *the universality principle*. It is so called because it states that the universe of data available from the information system should be available to be included by a designer in a view.
2. A logically independent view should be able to be structured, queried and manipulated as if it were an independent information system. This is sometimes referred to as *the modularity principle* because it states that the view mechanism should produce modules which encapsulate their definition. A user should not be able to distinguish between a module which is an entire information system, and one which is a view of a part of an information system.

Both principles are statements of the power of the view mechanism: System designers should be able to construct a view using any of the data from the information system, arranged in any logical arrangement, and manipulable by a user exactly as if the view were an entire information system.

In terms of the technique that we use, a view should be an arbitrary specified schema (thus an arbitrarily structured information system) which inherits its data from any legitimate queries carried out on the viewed system. Thus we define:

Definition 4 Let $\mathbb{E}$ be a schema. A *view* of $\mathbb{E}$ is a schema $\mathbb{K}$ and a schema morphism $k : \mathbb{K} \longrightarrow Q(\mathbb{E})$.

The idea is that $\mathbb{K}$ is an arbitrary schema, and it tells us how the view is to be structured as a (virtual) information system, and the schema map k indicates for each node in $\mathbb{K}$ how to find the corresponding data from the information system $\mathbb{E}$. The virtual information system can be structured in any reasonable way, and looks like an ordinary information system (modularity), and it can incorporate data obtained from any query applied to the information system with schema $\mathbb{E}$ (universality).

Naturally the mere existence of k does put some restrictions on the possible structures for the view $\mathbb{K}$. After all a schema map has to carry constraints to constraints. Thus, $\mathbb{K}$ cannot include constraints that have no corresponding constraints in $\mathbb{E}$. But this is no real limitation — of course it isn't possible to require constraints in the view which are not required in $\mathbb{E}$ since the data from $\mathbb{E}$, which might violate those constraints, is just inherited by the view.

Example 5 Returning to Figure 1 we can consider some simple examples of views. The simplest examples of views just include some of the data from the information system. Thus the Sales Department might have a view which includes only Customer, Product, Order, Location, Price and the relationships between them. Even more simply the Software Development Department might have a view of Figure 1 which only includes Virtual product. More interestingly the administrator of a particular warehouse might have a view of Figure 1 which consists of those Products which are located in his or her warehouse, those Deliveries which are scheduled for any of those products, and the Customers who are awaiting those deliveries. Notice that the nodes for the warehouse administrator's data are not present in Figure 1 but rather occur in the classifying category of the schema as pullbacks of Product and of the categorical product of Delivery and Customer over the warehouse location, or equivalently as `select` and `project` queries including a `where` clause specifying the administrator's warehouse. And of course the administrator's warehouse doesn't appear explicitly in his or her view — it is just used in the calculation of that view.

In fact, these examples are slightly oversimplified because attributes have been left out of Figure 1 and they add considerable complexity to views. For example, even the view for the Software Development Department might include or not include each of the attributes of Product and also each of the attributes of Location.

4. BIDIRECTIONAL LINKS

Now that we have a precise definition of view we are in a position to develop links as proposed in Section 1: Given two project data models which we seek to link we will develop a single schema $\mathbb{K}$ together with two schema morphisms which exhibit $\mathbb{K}$ as a view of each of the project data models.

The schema $\mathbb{K}$ is a single formal representation of those parts of the project data models that share common data in whatever form. Thus it might seem that we have achieved our goal since we have identified the common data that needs to be linked, and we have through the schema morphisms identified how that data can be obtained from the project data models. However, thus far we have only a one-way transmission of data "outward" from the project data models to the link schema $\mathbb{K}$. This is insufficient for a link between the project data models since when the common data in one of the models changes, we need it to change in the other project data model. Sure enough the corresponding data in the link model $\mathbb{K}$ will change for formal mathematical reasons [4], but without "inward" data transmission from $\mathbb{K}$ to the second project data model there can be no true link.

Therefore, we consider "inward" transmissibility of data changes. For precision, suppose K is the current state of a link model with schema $\mathbb{K}$. If K is a logically independent view of an information system state E, we should be able to insert and delete in K. But in reality K is a view of some of the data derivable from E, so an insert or delete in K must transmit some change to the underlying database state E. This change of the data of E must be arranged so that the change of the *view* of the data corresponds to the intended insert or delete.

Unfortunately, not all view inserts and deletes are transmissible in this way. This is the *view update problem* [5, Chapter 8] (or [6] but note that this later version has a much less detailed treatment).

Here is a minimal but real example of the difficulty. Suppose that $\mathbb{E}$ is the schema $\mathsf{Product} \longrightarrow \mathsf{Location}$ with no constraints. Let $\mathbb{K}$ be $\mathsf{Product}$ (with k the evident inclusion). An insert into $\mathsf{Product}$ in $\mathbb{K}$ is straightforward. But upon transmission to E we find that we cannot decide how to make such an insertion, because we need to specify the product's location too.

That example shows that we sometimes do not know how to transmit data inwards because there are too many choices. There are also examples where there is no possible state E of the project data model which could correspond to the effect of a specific insert in K — there are no choices.

Once again there has been controversy over the view update problem, arguably for the same reason as the controversy over views: People have defined updatability based on what their systems have been implemented to do, rather than seeking a fundamental definition. In fact there is a real difficulty with obtaining a fundamental definition in the usual framework because traditionally people have sought to define updatability for schema. Instead updatability, renamed in this context transmissibility, can be treated as a property of the current *state* of the information system, and transmissible can be defined as follows (based on [10]).

Definition 6 Suppose we have a view $k : \mathbb{K} \longrightarrow Q(\mathbb{E})$ and a current state E for the database corresponding to $\mathbb{E}$. A view insert/delete is *transmissible* if there exists a unique minimal insert/delete in E which achieves the view insert/delete. More precisely a view insert (delete) is transmissible if, among all of the inserts $D \rightarrowtail D''$ $(D'' \rightarrowtail D)$ which achieve the view insert, there is one $D \rightarrowtail D'$ $(D' \rightarrowtail D)$ which is initial (terminal) in the full subcategory of the slice category under (over) D.

Notice that we do not require a correspondence between the instances inserted into a view, and those inserted into E as a result of the transmission. To transmit a single inserted instance in a view we may need to carry out multiple inserts in various parts of E. We require only that the needed inserts are well-defined in the sense that as a collection they are the unique minimal insert in E which will achieve the intended view insert.

We move now to how we can use transmissibility to support the interaction of project data models along links specified by views.

5. CONSTRUCTING ENTERPRISE INFORMATION SYSTEMS

We now seek to apply the theory developed so far to the construction of enterprise information models from project data models. For simplicity consider two project data models with schemata $\mathbb{E}$ and $\mathbb{F}$.

We may assume that the project data models include some common data (otherwise we can construct the enterprise data model by taking the disjoint union of the project data models). However, it is unlikely that the common data will have the same names, or even share similar data structures. For example, one system might have separate nodes for each of several different types of product, while the other has a single node for all products. Or one system might only store the products which are at a particular location, while the other stores in one node the products from all locations and has an attribute to record their locations.

So we use logical data independence.

In Section 3 we defined a view to be a schema morphism $k : \mathbb{K} \longrightarrow Q(\mathbb{E})$. Furthermore we noted that $Q(\mathbb{E})$ contains all queries that can be performed on $\mathbb{E}$. So we can construct views that see the coproduct of all of the different types of product, or the subtype of products at all locations obtained by pulling back over the particular location, etc.

In this manner we construct a logically independent structure, a view, which contains matchable nodes from each of $Q(\mathbb{E})$ and $Q(\mathbb{F})$, and schema maps

$$Q(\mathbb{E}) \longleftarrow \mathbb{K} \longrightarrow Q(\mathbb{F}).$$

Next we explore inward state-based transmissibility for each schema map. Consider arbitrary states E and F of $\mathbb{E}$ and $\mathbb{F}$. An insert or delete in E affects directly the state K of $\mathbb{K}$ under the view k. We explore the transmissibility of this change of the state of K to the state F. Frequently we can prove lemmas that assure us that all such changes are transmissible [10]. In that case we have achieved interoperability, and the schema maps, expressed in say SQL terms, together with the construction used in the proof of the lemmas, provide the algorithms for the interoperability mechanism.

In some cases the failure to prove transmissibility identifies specific cases where interoperability fails, and these can be the subject of negotiations between the divisions involved to determine business rules or model changes that will assure good links with effective interoperability.

6. LINKING FIGURE 1

To provide a more detailed example, we now explore how this linking process could arrive at Figure 1.

We suppose that there are several divisions of the e-commerce organisation and that each has developed its own project data model.

As in Example 5 we may have a Sales Department, a collection of warehouse administrators (one for each warehouse), and a Software Development Department. In addition we would expect to have a Delivery Department which can see the entities Delivery, Product, Location, Order and Customer. And perhaps there are other departments too.

We construct Figure 1 incrementally, incorporating constraints as well as graphs as we link project data models pairwise (of course we could do several at once, but it is easier to explain pairwise linkages).

We can begin by linking the Sales Department's project data model with a project data model belonging to one of the warehouse administrators. Both models include an entity called Product but they are not semantically equivalent because the Sales Department's Products include all of the products that the organisation sells, while the warehouse administrator's Products are merely those products which occur in his or her warehouse. So we seek a common view. Since the warehouse administrator's view includes details about deliveries which are invisible to the Sales Department the common view will have to be a part of the warehouse administrator's project data model. In fact his or her Product, taken as a view of some of the project data model, can also be a view of the Sales Department's project data model by identifying it, via a schema morphism, with the pullback of the Sales Department's product over the warehouse's location (equivalently by the query `select product where location = warehouse-location`).

Customer needs to be treated in a similar fashion.

Similarly the other warehouses can be linked with the resulting model, one at a time.

Interestingly, once all of the warehouses have been linked all of the data is in place for the Delivery Department, but the deliveries corresponding to each warehouse were disjoint from those for the other warehouses, and at this point in the development there are separate delivery entities for each warehouse's products. So, to match the semantics of the Delivery entity for the Delivery Department with something in the cumulative model we need take the view of the cumulative model which exhibits all of the warehouses' deliveries as a single coproduct (disjoint union).

Fortunately Order and Customer have identical meaning between the cumulative model and the Delivery Department's project model, so they may be matched immediately.

But there is yet one more twist: The Delivery Department's Product entity only knows about physical products (because virtual products are delivered electronically). So the common view which amalgamates warehouse deliveries, and preserves customers and orders, needs to link the Delivery Department's Product with the cumulative model's query (another pullback) `select from product where location = location1 or location2 or ...` where each of the locations specified is a Physical location.

Finally the Software Development Department is linked similarly.

Of course at each step it is necessary to prove that view inserts and deletes are transmissible, but fortunately for our example this follows from propositions proved in [10], and the proofs of those propositions provide the linking code.

Note that the checks for transmissibility are vital, and if for example the Delivery Department did not include the Location of products in its project data model it would not be possible to transmit updates from the coproduct of the warehouses' deliveries to the individual warehouses' project data models.

This would be an example of a need for a negotiation between departments to obtain linkable project data models. Still, in this case it's rather unlikely that that difficulty would arise as it would correspond to the Delivery Department planning the deliveries of products without knowing where they were located!

7. CONCLUSION

This paper has presented a technique for the specification of links between project data models. In essence the technique aids in constructing enterprise models or enterprise information systems by linking project data models.

The technique is founded on a category theoretic data model which has already been used extensively in industrial consultancies. This foundation provides rich representations which aid in the design of interoperable views, and a well-understood mathematical basis which supports both constraint definition and the analysis required to guarantee the transmissibility of updates.

The approach uses fundamentally logical data independence in the new category theoretic sense. We use logical data independence between mathematically specified project data models to link data in those models even when those data are stored in perhaps radically different forms.

The interoperations themselves depend upon the transmissibility of updates between logically independent views. For this use of logical data independence to work it is necessary to mathematically analyse the logically independent views to ensure the required transmissibility.

The techniques described here are currently being tested on the interoperation of very large project data models in industry. Preliminary results have shown the techniques to be very successful, and the development of these case studies (and their important practical use in the organisations) continues.

One interesting limitation that we have found in applying the techniques in industry is that they seek to ensure the transmissibility of all updates in both directions (from project model E to project model F and vice versa). In fact, most interoperating systems we have investigated do need to interoperate in both directions, but updates of many particular data structures only occur in one direction (for example, a branch office changes orders and this needs to be indicated on the central warehouse systems, while the warehouse changes internal charges to accounts and this needs to be indicated on the branch office's systems). Using data communications terms, simplex interoperations (which correspond to the client-server model) are well understood, full-duplex interoperations (as described in this paper) are proving very useful but sometimes require more interoperability than is actually used, and half-duplex interoperations would seem to suffice. We are currently attempting to extend the techniques presented here to support "half-duplex" interoperations.

8 ACKNOWLEDGEMENTS

The research reported here has been supported in part by the Australian Research Council and the NSW Department of Health.

References

[1] A. Asperti and G. Longo. *Categories, Types and Structures: An introduction to category theory for the working computer scientist.* MIT Press, 1991.

[2] M. Barr and C. Wells. *Category theory for computing science.* Prentice-Hall, second edition, 1995.

[3] 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.

[4] C. N. G. Dampney, Michael Johnson, and Robert Rosebrugh. View Updates in a Semantic Data Model Paradigm. Proceedings of ADC2001, IEEE Press, 29–36, 2001.

[5] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, fourth edition, 1986.

[6] C. J. Date. *An Introduction to Database Systems.* Addison-Wesley, seventh edition, 2000.

[7] 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.

[8] J. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright. Initial algebra semantics and continuous algebras. *JACM*, 24:68–95, 1977.

[9] Michael Johnson and Robert Rosebrugh. Database interoperability through state based logical data independence. *International Journal of Computer Applications in Technology*, in press, 2001.

[10] Michael Johnson and Robert Rosebrugh. View updatability based on the models of a formal specification. Proceedings of Formal Methods Europe, LNCS 2021, 534–549, 2001.

[11] Saunders Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5, Springer Verlag, 1971.

[12] D. Scott. Continuous lattices. *Lecture notes in Mathematics*, 274:97–136, 1972.

[13] D. Scott. Domains for denotational semantics. *Lecture notes in Computer Science*, 140:577–613, 1982.

[14] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Volume 1, Computer Science Press, 1988.

[15] R. F. C. Walters. *Categories and Computer Science*. Cambridge University Press, 1991.