Reverse Engineering Legacy Information Systems for Internet Based Interoperation

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

Robert Rosebrugh
Mathematics and Computer Science
Mount Allison University
NB, Canada
Email: rrosebrugh@mta.ca

Abstract

The maintenance of legacy information systems is becoming increasingly common as needs for internet based interoperation drive system changes. This paper outlines new techniques for achieving interoperability among legacy information systems, usually without making major changes to the legacy code. The techniques involved use a limited type of reverse engineering to establish a formal model of relevant parts of the legacy systems, and they use existing interfaces to communicate between the code written to support the reverse engineered models and the legacy code. Interoperation is then achieved using mathematical techniques to support common logically data independent views of the reverse engineered models. The paper is somewhat theoretical as it provides a detailed exposition of the new techniques, but the techniques themselves are currently being tested in industrial applications with considerable success, and they are based on a framework which has been used in a number of major consultancies.
Keywords: Logical data independence, category theory, reverse engineering, legacy systems, databases, semantic data modelling

1 Introduction

The growing pervasiveness of internet technologies is driving business process changes that present significant software maintenance problems. In particular, legacy information systems need to be modified to support interbusiness and interdivisional interoperation. Frequently legacy systems have only limited documentation and major maintenance can involve significant reverse engineering.

This paper has grown out of the authors' previous work [16], [17] (which has been successfully utilised in a number of large scale consultancies including [7] and [26] — in fact the technique grew out of the needs of a very large information system specification consultancy [5]). Other related work includes [2], [11], [12], [15], [22], [23], [20]. Our work had been intended to limit reverse engineering problems in the future by improving specification techniques in the present. However, during the course of recent major consultancy work with our colleague Dampney [6] we have discovered that our techniques can be used for a form of limited reverse engineering that is sufficiently powerful to enable the design, implementation and maintenance of complicated information system internet based interoperations.

We use techniques that are drawn from *category theory*. Category theory ([3], [21], [28]) is a branch of mathematics that has been widely applied to specification in computer science. Examples of this application include: abstract data types [13], [14], semantics of programming languages [24], [25], and functional programming [1]. The techniques of category theory are graphical and based on a simple axiom set. They are highly valued for powerful semantic expressiveness.

We make extensive use of the idea of *logical data independence* (see, for example, [27]). Logical data independence supports views of systems' data that have different logical structure from the original systems. We argue that this is especially important in developing interoperations between legacy systems as it is rare that such systems would include common data in the *same* logical structure. We present below a mathematical treatment of logical data independence in a category theoretic framework and show how it is used to support our interoperation technique.

In summary our technique is as follows. Given two (or more) legacy information systems for which we wish to design interoperations, we use category theoretic specification techniques to develop logically data independent models of (parts of) the legacy information systems. These models are formal representations of reverse engineered subsystems. We then develop a single logically data independent model of (parts of) each of the reverse engineered subsystems. This model, $\mathbb{K}$, is the locus of interoperation. It is a single formal representation of those parts of the original systems that will be involved in the interoperations. It remains to develop the interoperation code.

To develop the interoperation code we need to explore the *propagatability* of update information from each legacy system, "out" to the common submodel $\mathbb{K}$ and back "in" to the other information system(s). It turns out that the outward propagatability (from the legacy system to $\mathbb{K}$) is guaranteed, first by the construction of the reverse engineered model, and then for formal mathematical reasons from the reverse engineered model to $\mathbb{K}$. Inward propagatability on the other hand *cannot* be guaranteed. Instead, inward propagatability needs to be explored in two steps. First from $\mathbb{K}$ to the reverse engineered submodel we use mathematical methods (since both models are formally specified), and a catalogue of some of these methods has been published in [17]. Then model checking techniques are used to test inward propagatability from the reverse engineered submodel to the legacy system (in fact these checks are typically part of the reverse engineering process we use, and there is no need to implement code for this part of the inward propagation since the legacy system interface is used to carry it out).

The plan of the paper is as follows. In Section 2 we outline the category theoretic information system specification techniques that we use. In Section 3 we develop a treatment of logical data independence in our category theoretic framework. Section 4 explores the propagation of updates across data independence boundaries. Finally in Section 5 we show how the theory developed in the earlier sections can be used to link interacting legacy information systems using limited reverse engineering.

2 Information System Schemata and States

In this section we provide a brief introduction to category theoretic information system specification. For more detail we refer the reader to [16]. Any introduction to category theory (for example, [3] or [28]) contains the definitions and elementary properties of the category theoretic concepts needed below including commuting diagrams, limits, and coproducts. We will assume some familiarity with the terminology of Entity-Relationship (ER) model [4].

We have called our category theoretic data model the *sketch data model* since it relies on the syntactic specification device known as a mixed sketch [3]. An *information system schema* or *EA sketch* is specified in two parts. The first requirement is a graph, roughly corresponding to an ER graph. The second element is a set of (categorical) constraints. The constraints take three forms:

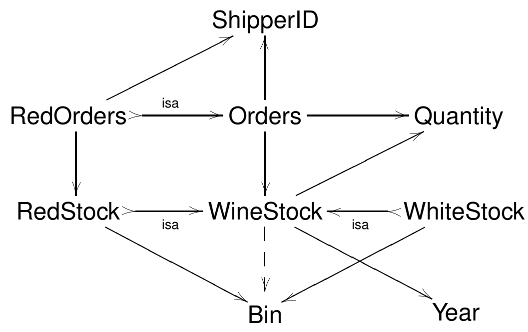

Figure 1. A fragment of a Winery data model schema $\mathbb{E}$

1. Commuting diagrams that are pairs of paths in the graph with common origin and destination.

2. Limit constraints that specify that a certain node in the graph is to act as the "limit" of a specified diagram in the graph.

3. Coproduct constraints that specify that a certain node in the graph is to act as the "coproduct" of specified nodes in the graph.

EA sketches are special cases of the more general mixed sketch. They have been studied by the authors and coworkers [18], [19].

An *information system state*, also called a database state or instance, is an assignment of (i) a finite set for every node in the schema (the set of instances or values of that entity or attribute), and (ii) 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). These assignments are required to satisfy:

1. The commuting diagrams do indeed commute as diagrams of corresponding functions.

2. The sets assigned to limit nodes are indeed the limits of the corresponding specified diagrams of functions.

3. The sets assigned to coproduct nodes are indeed coproducts (disjoint unions) of the sets assigned to the corresponding specified nodes.

In other words, a state for an information system schema is a diagram of sets and functions which is the same shape as the graph, and whose sets and functions satisfy the constraints. When the context is clear, we will refer to a database state D for a category theoretic data model schema $\mathbb{E}$ as simply a state D of $\mathbb{E}$.

Figure 1 is the graph part of an example information system schema. It is a small fragment of a business schema for a winery. The full schema is *much* bigger. The part shown includes among its constraints the requirements that: the upper left triangle commutes, the rectangle commutes and is a pullback (a kind of limit), and that the WineStock node is a the coproduct of RedStock and WhiteStock. Also shown are subtype relations denoted by $\rightarrowtail$: RedOrders is a subtype of Orders, RedStock and WhiteStock are subtypes of WineStock. In fact, it also follows from general properties of coproducts and pullbacks that the functions corresponding to these isa arrows are injective in any state of this schema.

The constraints ensure that: The ShipperID of a RedOrder is the ShipperID of the Order (the commuting triangle); Exactly those Orders which are for RedStockss are RedOrders (the pullback); All WineStocks are either RedStocks or WhiteStocks (the coproduct).

We note two further points about the schema. First, the dashed arrow from WineStock to Bin is *not* part of the schema. However, the coproduct constraint on WineStock and the arrows from RedStock and WhiteStock to Bin mean that in any state of the schema there is a uniquely defined function from the set of WineStocks to the set of Bins which agrees with the Bin assignments by color. Secondly, it is important to realize that not *all* pairs of paths need to be commutative diagrams. For example, the Quantity of an Order is different from that of the WineStock of the Order.

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

Incidentally, any category has an underlying schema. 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. When we refer to a schema map into a category (as we will in Section 3) we in fact mean a schema map into the underlying schema of the category. Moreover, an information system state is itself precisely the same thing as a schema map from the information system schema to the underlying schema of the category of finite sets.

A schema $\mathbb{E}$ generates a *classifying category* denoted $Q(\mathbb{E})$. 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 important technical uses, and, as we are about to see, is important for logical data independence too.

Figure 2 shows a part of the schema $\mathbb{E}$ from Figure 1 together with a part of the graph of $Q(\mathbb{E})$. The node 1,

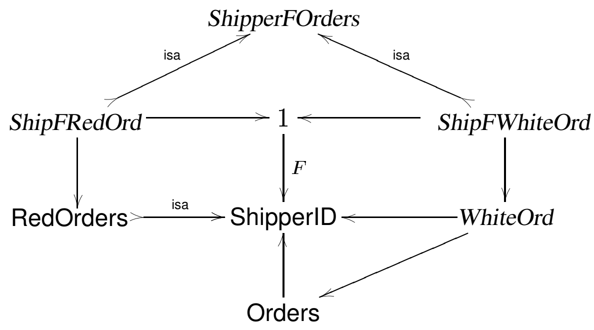

Figure 2. A fragment of a classifying data model schema

present in any EA schema, was not shown in Figure 1. F is the ID of a supplier F and is part of $\mathbb{E}$. The new nodes from $Q(\mathbb{E})$ (in italic) include *ShipFRedOrd*, constructed as a pullback and *ShipFOrders* constructed as a coproduct of *ShipFRedOrd* and *ShipFWhiteOrd* — the latter a pullback constructed using the new node *WhiteOrd*. The new nodes correspond to query results on the schema. The presence of these nodes is crucial to our description of views below.

3 Data Independence and Views

The implementation of physical data independence allows a user to work with a database without needing to concern themselves with how the data are physically stored. On the other hand, *Logical data independence* allows working with an information system without concern for how the data are logically arranged — a user is insulated from the schematic structure of the information system. With logical data independence a user can view and manipulate data in an arrangement, a logical data structure, which is independent of the actual logical structure of the system.

The importance of logical data independence is that it permits modification of an information system schema, perhaps adding more attributes, or extra entities and relations as a business evolves while continuing to use the same applications programs, and thus to present staff who do not need access to the new structures with the same interface that they were using before the change.

Most modern systems include the implementation of some form of view mechanism in order to obtain partial logical data independence. Unfortunately, the implemented view mechanisms suffer serious restrictions, both on the views that can be defined and further in how the viewed data can be manipulated.

True logical data independence should have the following properties:

1. A logically independent view should be able to contain any data that can be derived from the data stored in the underlying information system. We call this *the universality principle*. Of course particular views are sometimes constructed to *limit* access to data. A view can only see the data that it is designed to include. The point of the principle is that in designing a view we should be able to choose to include in that view any data from the universe of data available from the system.

2. A logically independent view should be able to be structured, queried and manipulated as if it were an independent information system. We call this *the modularity principle*. The point of the principle is that as users we should not be able to distinguish a logically independent view from an information system, and so we should for example be able to define logically independent views of our view, etc, and as designers we should be able to structure the data in the view in any manner (compatible with the underlying data) as if we were designing an independent system.

Interpretation of the two principles just defined must take place relative to the data model employed. With respect to the universality principle, the universe of data available from a system depends upon the queries that are available, that is, it depends upon the queries that are supported by the data model in use. In a similar way, the modularity principle assures us that we can structure viewed data as if it were an information system. The range of structures we have available for our systems depends entirely on the data model we are using.

We now present our approach to logical data independence in the sketch data model.

Definition 1 Let $\mathbb{E}$ be an information system schema and let $Q(\mathbb{E})$ be its classifying category. A *view* of $\mathbb{E}$ is a schema $\mathbb{K}$ and a schema morphism $k : \mathbb{K} \longrightarrow Q(\mathbb{E})$. For a given view, we will refer to the database state corresponding to the schema $\mathbb{E}$ as the *underlying database* of the view.

What makes such a morphism a view of an information system schema? Since $\mathbb{K}$ is itself a schema, it defines the structure of the view schema as a (virtual) database. The schema map k determines, for each node in $\mathbb{K}$, where to find the corresponding data in the state of the underlying database.

As an example of a view, we let $\mathbb{K}$ be the information schema in Figure 3 (*without* the constraint that ShipperFOrd is a coproduct), and k just the inclusion in $Q(\mathbb{E})$. One of the nodes of $\mathbb{K}$ is from $\mathbb{E}$ while others only exist in $Q(\mathbb{E})$.

To see that views support true logical data independence let us consider the principles above.

$$
\begin{tikzcd}
\mathit{ShipFRedOrd} \arrow[r, tail, "\text{isa}"] \arrow[d] & \mathit{ShipperFOrd} & \mathit{ShipFWhiteOrd} \arrow[l, tail, "\text{isa}"'] \\
\text{RedOrders} & &
\end{tikzcd}
$$

Figure 3. View schema for a view of the Winery schema

Universality: As we noted above and in more detail in [8], the classifying category $Q(\mathbb{E})$ has a node for every query that can be made on an information system with schema $\mathbb{E}$. Thus, when designing a view we can choose nodes whose value under k, that is, whose underlying data, arises from *any* query on the underlying database state.

Modularity: In the sketch data model an information system schema is presented by giving the graph and constraints. Now $\mathbb{K}$ which is the view schema, can itself be first structured, and then queried, or manipulated, as if it were an ordinary database. Nevertheless it must be remembered that the data available in a state for $\mathbb{K}$ have come from the underlying database state of the schema $\mathbb{E}$, and so they will always be subject to the constraints in $\mathbb{E}$. The implications of this will be taken up in the next section.

It is important to remember that the schema map $k : \mathbb{K} \longrightarrow Q(\mathbb{E})$ defining a view carries constraints to constraints. Thus, the schema $\mathbb{K}$ cannot include constraints that do not map to corresponding constraints in $\mathbb{E}$. This certainly matches our intuition about how views should operate — it should be impossible to require constraints in the view schema that are not required in the underlying database schema since the data from a state of the underlying database, which might violate those constraints, is exactly the data seen by the view. However, this does not violate modularity. Rather, it clarifies the meaning of "compatible with the underlying data" in the statement of the modularity principle.

It should also be noted that the constraints just mentioned have nothing to do with the queries used to define the view. A view of the orders from shipper F does not include a constraint that says orders come only from shipper F. Instead it takes its values from queries of the form

```
select  ...  where SHIPPER = F.
```

4 Propagatability and View Updates

In this section we consider propagatability of view updates. Clarity on this will permit consideration of information system interoperability. If K is a state of $\mathbb{K}$ for a logically independent view k of an information system schema $\mathbb{E}$, we should be able to insert and delete in K. However k

is a view of some of the data (derivable) from $\mathbb{E}$. Thus an insert or delete in a state K must transmit the change to the underlying database state D of $\mathbb{E}$. The change to the data of D must be made in such a way that the resulting *view* data corresponds to the intended insert or delete. Unfortunately, not all view inserts and deletes are propagatable in this way.

We consider some examples. Suppose that the graph of $\mathbb{K}$ is a part of that for $\mathbb{E}$, but that $\mathbb{E}$ includes an extra constraint that is not required in $\mathbb{K}$ (remember that each constraint in $\mathbb{K}$ is mapped by k to a constraint in $\mathbb{E}$, though nothing prohibits extra constraints in $\mathbb{E}$). An insert in the view schema state which is in violation of the extra constraint is acceptable to the (virtual) view database. However, the insert cannot be propagated to the underlying database (because it violates the extra constraint). In the example view of Figure 3, the view does not include the coproduct constraint on *ShipperFOrd* so it would be possible to insert an item in the value of *ShipperFOrd* without adding an item to either of its summands. This would violate the constraint in $\mathbb{E}$.

As a second example consider the schema

$$\mathsf{Age} \longleftarrow \mathsf{Person} \longrightarrow \mathsf{Address}$$

with no constraints. Let $\mathbb{K}$ be $\mathsf{Person} \longrightarrow \mathsf{Address}$ (with k the evident inclusion). An insert into Person in $\mathbb{K}$ requires the specification of the person's address. But upon propagation to D we find that we cannot make such an insertion, because we need to specify the person's age too.

Are these problems violations of the modularity principle? It could be argued that they are. But we point out that the view schemata are still independent information systems. As a result of the view morphism they contain extra "hidden" constraints which are not true violations of modularity. Such hidden constraints include the "extra" constraint, in the first example and a constraint prohibiting inserts into Person in the second example.

Taking this point of view, we need to characterize the hidden constraints. What inserts and deletes are propagatable from views to their underlying databases? This is known as the *view update problem* [10, Chapter 9]. The view update problem has provoked widely varying attempts at solution and some confusion because workers have sought to answer it in terms of schemata. Our position is that view updatability is determined by the current *state* of a database, and we define propagatable as follows.

Definition 2 Suppose that we have a view $k : \mathbb{K} \longrightarrow Q(\mathbb{E})$ and a current state D for the database corresponding to $\mathbb{E}$. A view insert is *propagatable* if there is a unique minimal insert in D (the underlying database) which achieves the view insert.

There is a need for precision about the "unique minimal insert" in the definition. The phrase is easily misunderstood. In fact it means that among all of the inserts $D \rightarrowtail D''$ which achieve the view insert, there is one $D \rightarrowtail D'$ which is initial in the full subcategory of the slice category under D ([3]). The definition of *propagatable delete* is similar using "unique minimal delete".

For an example of a propagatable delete, consider the view in Figure 3. Suppose that an item in *ShipFWhiteOrd* and its image in *ShipperFOrd* are deleted in the view state (both deletes are needed to avoid violating $Q(\mathbb{E})$ constraints). These deletes require a deletion in the value of *WhiteOrd* (in $Q(\mathbb{E})$) and then in Orders. They can be done in a unique correct way with only single deletions while maintaining the constraints. The resulting state of the schema $\mathbb{E}$ satisfies the definition.

The definitions can be unified and extended with the mathematical concept of *fibration*. General propagatability results for particular schema shapes have been studied in [18], but we will not need them here. We are ready to show how state-based propagatability and logical data independence can support interoperability for reverse engineered legacy information systems.

5 Reverse Engineering Application

The setting for our application requires two or more systems with a need to interoperate. The systems could require business-to-business web-based transactions, or simply collaborative work among divisions inside one organization.

We begin with a limited reverse engineering approach. Presumably the two systems have some common data that we seek to synchronize. Probably those data are stored in radically different ways. We aim to develop models of each of the systems which include all of the data that are likely to be relevant for interoperability, and as many of the constraints as we can discover. Useful tools of course include any documentation that is available, most especially data dictionaries as a complete list of attributes is very useful. However, the main tool is the legacy system itself which can be explored using its own interface. One of the great advantages of the methodology below is that it interacts with the systems through their extant interfaces as far as possible, and in fact aspects of the system which are not revealed through interaction with the interface do *not* need to be modelled.

To our surprise, there is a very large body of methodology that can be used to aid in the reverse engineering process. It turns out that our need to elicit structure and constraints from the legacy systems to incorporate in our models corresponds exactly with the needs when modelling real world and business systems in order to design information systems. Thus the whole body of information system specification methodology, and particularly the category theoretic specification methodology, can be used to try to capture a reasonable formal representation of a legacy system. Furthermore, the fact that we only need to model a small part of each legacy system to support interoperation makes the elicitation task relatively straightforward.

The use of the extant methodologies does not make the reverse engineering process trivial, but it is enormously easier than most reverse engineering tasks. Nevertheless, it is important not to understate its importance: Many information systems development processes aim to develop detailed documentation, in stark contrast to what is available for many legacy systems, because the designers expect to work with a full (often formal) specification for major maintenance tasks, and view reverse engineering as so difficult that it is better to reimplement systems than to try to understand them well enough to modify them. Yet our comparatively easy reverse engineering process in fact provides the information needed to develop quite complicated interoperating systems.

Throughout the remainder of this section we will assume that the systems have been reverse engineered to give a representation of relevant parts of them which is based on the sketch data model. Thus we have a rich description of the data via schemata which include constraints. We note that it is a logically data independent description since we have structured the data in the reverse engineering process in whatever manner seemed most convenient, and we do not need to be concerned about the actual logical structure used in the legacy system.

Interoperations among the models must be developed for extension to the legacy systems using their interfaces.

Suppose two systems which need to interoperate have reverse engineered schemata $\mathbb{E}$ and $\mathbb{F}$, with current states E and F respectively. For the following paragraphs the reader might think of $\mathbb{E}$ as the schema we considered in Figure 1. For $\mathbb{F}$ one might consider the schema for a shipper (see Figure 4). We will pursue the example below.

Of course, the information systems must include some common data (otherwise we would not need them to interoperate) but it is unlikely that the common data will have the same names, or even share similar data structures in the legacy systems. For example, one system might have separate nodes for each of several different customers, while the other has a single node for all customers (as with Orders in $\mathbb{E}$ and *ShipperFOrd* in $\mathbb{K}$ above). 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 again use logical data independence.

Recall that a view consists of $k : \mathbb{K} \longrightarrow Q(\mathbb{E})$, and 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 customers, or the subtype of products at all locations obtained by pulling back over the particular

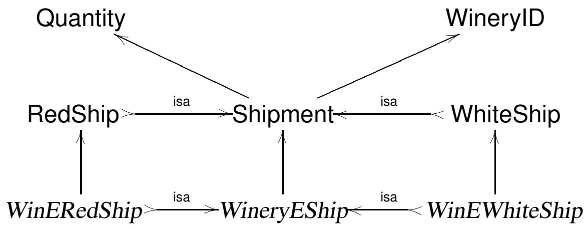

Figure 4. A fragment of $Q(\mathbb{F})$, a data model schema for a Shipper

location, that is, the result of a query of the form

```
select  ...  where LOCATION = ...
```

Our objective is the construction of a logically independent structure, namely a schema $\mathbb{K}$. The schema should contain matchable nodes from each of $Q(\mathbb{E})$ and $Q(\mathbb{F})$ so that we can construct schema maps (views)

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

Consider the schema $Q(\mathbb{F})$ in Figure 4. It represents a fragment of the classifying category for a shipper's data model $\mathbb{F}$ (with the nodes in $\mathbb{F}$ itself distinguished by font). As an exercise, the reader could fill in the constraints and additional arrows necessary to make Winery $\mathbb{E}$'s shipments arise as pullbacks.

Suppose we simplify the schema $\mathbb{K}$ of Figure 3 by deleting the node RedOrders and continue to denote the obvious view $k : \mathbb{K} \longrightarrow Q(\mathbb{E})$. The view $l : \mathbb{K} \longrightarrow Q(\mathbb{F})$ is obtained by assigning the nodes in the bottom row of Figure 4 to nodes of (the simplified) Figure 3.

It is now possible to consider state-based propagatability across this structure (called a *span*) of schema maps. Consider arbitrary legal states E and F of $\mathbb{E}$ and $\mathbb{F}$. An insert or delete in F affects directly the state K of $\mathbb{K}$ under the view l. We consider the propagatability of this change of the state of K to the state E via the view k.

For example, suppose that an item in the value for F of *WhiteShip* and its corresponding *Shipment* are deleted. If it happens to be an item in the value of *WinEWhiteShip* then according to the view l there is a delete in the state of $\mathbb{K}$. As we discussed above, this delete is propagatable.

Frequently we can prove lemmas that assure us that all such changes are propagatable [17]. 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 cases where propagatability fails in general, we may identify specific cases causing this. They can be the subject of consultations between the organizations involved to determine constraint or system changes that will assure interoperability. Unfortunately, when system changes are agreed upon, we usually need to resort to more traditional maintenance techniques. The advantage is that the precise issues that need to be corrected have then been identified and agreed upon.

A straightforward extension of the methodology is appropriate when organizations wish to merge operations. What is required is that each organization's schema is extended in harmonization with that of the other in its reverse engineered form. This will allow the common view $\mathbb{K}$ to be "widened" to include the resulting larger parts of the individual organizations' schemata. If the organizations with schemata $\mathbb{E}$ and $\mathbb{F}$ above were to merge operations, it would be appropriate to extend the schema for each to include (at least an expanded part) of the other's schema. This would allow an expansion of $\mathbb{K}$ to include, for example, information on *all* Shipments and perhaps all WineStocks.

6 Conclusion

This paper has developed an approach to the maintenance of legacy information systems. One of the principal problems which has arisen with the growing use of internet technologies to link business systems is the need for extant systems to be modified so as to support internet based interoperations.

The approach presented here is founded on the sketch data model which has already been widely tested in industrial consultancies. This foundation provides relatively rich representations which aid in properly designing interoperable views, and a well-understood mathematical basis which supports both constraint definition and the analysis required to guarantee the propagatability of updates.

The approach uses logical data independence in two senses.

First it develops reverse engineered subsystems and presents them as logically independent formal models of (parts of) the legacy systems. This is a very limited form of reverse engineering (and is correspondingly easier to achieve), and the logical data independence ensures that it is not necessary to completely match the internal data structures of the legacy systems. The veracity of the models is ascertained by model checking. The completeness of the models is not important as we only seek to reverse engineer enough of the legacy system to support the planned interoperations.

Secondly the approach uses logical data independence between mathematically specified models to connect data in the reverse engineered submodels even when those data are stored in perhaps radically different forms.

The interoperations themselves depend upon the propagatability of updates between logically data independent systems. For the first use of logical data independence (the reverse engineering part) propagatability is tested as the reverse engineered model is developed, and is then guaranteed. For the second use of logical data independence it is necessary to mathematically analyze the logically data independent views to ensure the required propagatability.

The techniques described here are currently being tested on the interoperation of the very large Department of Health data models [6]. Preliminary results have shown the techniques to be very successful, and the development of this case study (and its important practical use in the organization) continues.

One interesting limitation that we have found in applying the techniques in industry is that they seek to ensure the propagatability of all updates in both directions (from legacy system A to legacy system B 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, the customer changes orders and this needs to be indicated on the supplier's systems, while the supplier changes invoices and this needs to be indicated on the customer'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.

7 Acknowledgements

The research reported here has been supported in part by the Australian Research Council, the Canadian NSERC, and the Oxford Computing Laboratory.

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] K. Baclawski, D. Simovici and W. White. A categorical approach to database semantics. *Mathematical Structures in Computer Science*, 4:147–183, 1994.

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

[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] C. N. G. Dampney and Michael Johnson. TIME Compliant Corporate Data Model Validation. Consultants' report to Telecom Australia, 1991.

[6] C. N. G. Dampney and Michael Johnson. Fibrations and the DoH Data Model. Consultants' report to NSW Department of Health, 1999.

[7] C. N. G. Dampney, Michael Johnson and G. M. McGrath. Audit and Enhancement of the Caltex Information Strategy Planning (CISP) Project. Consultants' report to Caltex Oil Australia, 1993.

[8] 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.

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

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

[11] 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.

[12] Zinovy Diskin and Boris Cadish. Variable set semantics for generalised sketches: Why ER is more object oriented than OO. In *Data and Knowledge Engineering*, 2000.

[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 The Unified Computation Laboratory, eds Rattray and Clarke, *Springer Workshops in Computing*, 77–84, Springer-Verlag, 1994.

[17] Michael Johnson and Robert Rosebrugh. View updatability based on the models of a formal specification. *Proceedings of FME 2001*, 534-549, *Lecture Notes in Computer Science* 2021, 2001.

[18] Michael Johnson and Robert Rosebrugh. Universal view updatability. preprint, 2001.

[19] Michael Johnson, Robert Rosebrugh, and R. J. Wood. Entity-relationship models and sketches. Submitted to *Theory and Applications of Categories*, 2001.

[20] E. Lippe and A ter Hofstede. A category theoretical approach to conceptual data modelling. *RAIRO Theoretical Informatics and Applications*, 30:31–79, 1996.

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

[22] F. Piessens and Eric Steegmans. Categorical data specifications. *Theory and Applications of Categories*, 1:156–173, 1995.

[23] F. Piessens and Eric Steegmans. Selective Attribute Elimination for Categorical Data Specifications. Proceedings of the 6th International AMAST. Ed. Michael Johnson. *Lecture Notes Computer Science*, 1349:424-436, 1997.

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

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

[26] 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*, 55:33–46, Elsevier Science, 1999.

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

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