

Extended Abstract of Data, Schema and Ontology Integration

Joseph A. Goguen
Univ. of California, San Diego
Dept. Computer Science & Engineering

1 Introduction and Motivation

Data integration is emerging as a major challenge in the early 21st century. The rise of inexpensive storage media, data warehousing, and especially the web, have made available vast amounts of data. But unfortunately, it can be very difficult to find what you want, and to combine it properly to get what you need. Reasons for this difficulty include the highly variable structure and quality of data; for example, science labs and businesses often have data stored in spreadsheets, or even just formatted files, with little or no “meta-data” to document either format or meaning; moreover, some entries may be incomplete, corrupted, or inconsistent. If all documents had associated “schemas” (also called “data models”) that accurately described their structure, and if fully automatic schema integration were feasible, then it would be possible to solve several interesting problems at the syntactic level [4]; however, these two assumptions are far from true. Moreover, format is only a small part of the difficulty, most of which is semantic and pragmatic, not syntactic. Although the so-called “semantic web” vision for the world wide web has received the most publicity, similar problems have appeared, and are being confronted, in other domains, including “workflows” to automate processing the enormous datasets that are increasingly common, not just in astrophysics, proteomics, high energy physics, etc., but also in ecology, agriculture, pharmacology, e-business, geology, and numerous other areas. The growing popularity of XML may make things easier, but it cannot solve the basic problems.

So called “ontologies” have been proposed as a solution. These are not philosophical (metaphysical) assertions about basic “world substances,” but terminological systems, items from which can be attached to items in e-documents. They cannot capture real world semantics, but only logical relations between predicates, such as that all humans are mammals; the actual meanings of “human” and “mammal” remain unformalized. Moreover, a given domain may have competing ontologies, each in some ways incomplete and/or ambiguous, and possibly written in different ontology languages, which in turn may be based upon different logical systems. OWL and RDF are currently most prominent, but others include Ontologic, *ALC*, KIF, KL-ONE, XSB, Flora, and OIL; specialized ontology languages, e.g., Ecolingua and EML for ecology, tend not to have a formal semantics. It follows from this that the ontology approach to data integration may require not just schema and ontology integration, but also ontology language integration, and even ontology logic integration, in such a way that semantics is respected throughout the entire “integration chain,” from actual datasets or “documents,” through schemas and ontologies, up to ontology logics. Institutions and their morphisms are promising for understanding the meaning of this integration chain, untangling some problems involved, and even suggesting some sound solutions; however, it should be noted that the deep difficulties of data integration have not only technical, but also social aspects.

Section 2 below provides informal background on databases, schemas, schema mappings, ontologies, workflows, and their role in data integration, written for logicians and others not especially familiar with current database technology. Section 3 describes the author’s recent work on database theory and practice, including abstract schemas with constraints and their morphisms for *m-to-n* matches with semantic functions and conditions; this provides a new theory for integrating data in diverse formats. Section 3.1 briefly describes our tool SCIA, which implements this theory for data that has XML DTDs or SML Schemas. Section 4 sketches our approach to data integration via ontology integration, using prior work on institutions, institution morphisms, and Grothendieck institutions to formalize the logic level of the integration chain. Section 4 also includes a brief discussion of connections with the information flow (in the sense of Barwise and Seligman) approach to ontologies.

Acknowledgments I wish to thank Jenny Wang and Young-Kwang Nam for collaboration on the schema matching tool, Kai Lin and Vitaliy Zavesov for work on its implementation, and Bertram Ludäscher for valuable discussions. This material is based on work partially supported by the National Science Foundation under Grant No. ITR 0225676, the Science Environment for Ecological Knowledge (SEEK) project.

2 Background

Information comes from and goes to human beings: pixels, bits, marks on paper, etc. have no meaning in themselves, but must be interpreted. So what is stored in databases (or books, cave walls, DVDs, or other media) is not information, but data. Interpretation has been studied for a long time by numerous disciplines, including semiotics, hermeneutics, pattern recognition, literary criticism, ethnomethodology, statistics, media studies, machine learning, phenomenology, cognitive neuro-science, psychophysics, and more. What seems clear is that it is still poorly understood, in part because the narrow confines of the individual disciplines prevent a comprehensive perspective on a complex phenomenon that transcends such boundaries. It is distressingly common to sweep as much complexity as possible under the rug of “context,” and it is a pervasive error to think that all the contextual information needed to interpret data can be digitally encoded and mechanically applied. Human beings are the ground for all information and all interpretation, and human society is the matrix within which all meaning is embedded, and it is a side effect of that process to create a constantly shifting foreground and background, with the latter being called “context.”

Moving up the chain a bit, and confining attention to digital data, we find a great variety of storage media, including tape, CD, DVD, flash memory, hard and scuzzy disk, RAM, stick, jukebox, and more. This level is usually taken for granted, but it is non-trivial, as anyone who has ever had to deal with the internals of device drivers can attest. Data at this level is already structured into bits, bytes, tracks, and other device-dependent subdivisions, oriented towards particular patterns of use.

Databases further organize storage media to facilitate operations that either interrogate or update the contents. This organization may involve data encodings according to data type, e.g., integer, character, string, floating point, etc., as well as further structuring, to make relationships between different data items explicit. Types determine what operations are available on the underlying bits, and provide humans with valuable clues about interpretation, while structuring makes it easier to find related data, (usually) provides associated names, and again facilitates interpretation. Unfortunately, several different ways of structuring data, called **meta-models**, are in common use. One, called **relational**, structures data into relations with fields, while another, called **object oriented**, structures data into objects with inheritance and attributes. With the rise of the web, XML is poised to overtake these; its approach is called **semi-structured**, hinting at its greater flexibility. There are also legacy databases that use so-called hierarchical meta-models.

Data about data is called **meta-data**. The most important, or at least best understood, meta-data for a database is its **schema**, which describes its conventions for structuring, typing, and naming data. Schemas for relational and object oriented databases are well established and well studied, as they are for XML, with its DTDs and XML Schemas. However, a great deal of critically important meta-data falls outside the scope of schemas. For example, while it may be possible to say that a certain item of data is measured in feet, it is not possible to say what a “foot” actually is, or to relate it to other units, such as meters. In some cases, the way measurements are taken, the so-called “protocol,” can be very important. For example, in ecology, species density is defined as species count divided by area. But for marine species, volume may be the relevant denominator, and it will matter how species counts are obtained, e.g., by a net that is dragged for a certain amount of time, or by observation from some fixed point. The time of year and time of day may also be important, since species migrate at different times of the day and year, and of course variations in weather modify these patterns. The taxonomies used to classify species may also differ, e.g., different criteria may be used, different granularities of classes, etc. Different modes of observation may also have different inherent inaccuracies, some of which may be systematic, e.g., if certain colors are distorted or difficult to distinguish underwater. And all this is just one tip of an enormous iceberg of potentially critical information.

As the above discussion suggests, making use of data in one database may require integrating it with data from others. For example, to interpret a measurement of the density of gray whales at a certain time and place, we might want to know the ocean temperature, the path and time of migration for that species, and to compare current data with data from previous years, among other things. This requires knowing what factors are relevant, how important they are, where to get the necessary data, and how to process it. In general, the necessary data is stored in different databases at different sites, and although it might all be accessible over the internet, some of it may require passwords and/or special knowledge. Typically today, scientists import all the data they need into their own lab, massage it in various ways (such as averaging or interpolating time series to achieve compatability with other time series), and finally process the integrated data, often using hand coded *ad hoc* programs. All this is a far cry from writing a query in a standard language like SQL or XQuery, and running it over a single database. However, the goal of much current

research is to bring data integration closer to that paradigm; some of this research is described in the next three subsections. It is not claimed that these three areas include everything that is relevant, or that they are likely to be sufficient for solving all problems of data integration.

2.1 Schema Integration

One standard approach to data integration is to construct a “global” database connected by “views” to the various “local” databases such that the global database contains all the data of interest from the local databases, and can be queried to obtain exactly what is needed for some particular purpose. However, it is likely that the local databases are all changing over time, and it seems wasteful and inefficient to duplicate and continually update everything. So instead, the global database is only a “virtual” database, represented by a schema, and the views are actually schema mappings. Somewhat more formally, and using some language from category theory, views are schema morphisms, and the situation described above is a cone (and/or cocone) in the category of schemas, with apex a global schema G over local schema L_i . The case of a cone $v_i : G \rightarrow L_i$ is called “local as view” while the case of a cocone $u_i : L_i \rightarrow G$ is called “global as view.” The latter is adequate for processing queries, but the former is also needed if updates are to be processed.

Unfortunately, it can take a lot of effort to construct the necessary views, effort that is usually not worthwhile for any single project. Moreover, the local databases and their schemas are in general evolving, as are the needs of the domain of activity involved, so that additional ongoing work is needed to “maintain” the views, i.e., to keep them up to date; often, no single research project will have the resources or motivation to do this additional work. This motivates the construction of tools to automatically construct schema mappings. Although there has been a good deal of effort in this area, one outcome is that total automation seems infeasible, so that some human intervention is needed to achieve quality results; see Section 3.1.

A different kind of gap is that the local databases often use different meta-models, while the notion of view is limited to schemas having the same meta-model. An expensive option is to “wrap” a database using one meta-model with a “mediator” providing a schema with a different meta-model; this may be practical for legacy databases using obsolete models, but is otherwise not usually worth the trouble. This implies that we need a way of defining views between schemas having different meta-models. To avoid an *ad hoc* approach in which there are n^2 different notions of view among n different kinds of meta-model, it would make sense to have a single notion of schema that can be used for databases having any meta-model; then we only need one notion of schema morphism, which should specialize to views for the individual meta-models. Such a notion is that of an abstract schema, described in Section 3; as far as we know, this work provides the first semantics for n -to- m matches with semantic functions and conditions, among schemas that may have different meta-models.

2.2 Ontologies

Ontologies, in the sense discussed in Section 1, are used to express logical relations among predicates, and in general may involve constants and terms, as well as logical connective, quantifiers, etc. The intention is to establish a standard terminology, with logical relations among terms, for use over a set of databases, in order to simplify both searching for and integrating data. Ontologies are often given as sets of Horn clauses, but as already noted, there are a number of competing formalisms. Moreover, not all ontology languages have a formal semantics, and among those that do, there are great differences in expressiveness.

These problems are analogous to those already encountered at the meta-model level, as discussed in Section 2.1, and there is also an analogous way to solve them, namely to formalize logics and morphisms among them in a uniform way. While logicians have seemed reluctant to formalize the notion of “a logic,” computer scientists have been less shy, and there is now a considerable literature on “institutions” [10], which are an abstraction of satisfaction relations between sentences and models, in the style of Tarskian semantics, but parameterized over a category of signatures. A very high level of generality is achieved through the use of category theory. (It is unfortunate that the term “model” becomes so heavily overloaded in this collision of database theory and logic; the database notion that most closely corresponds to “model” in the sense of Tarskian model theory and institutional abstract logic is actually that of a database.)

2.3 Workflows

The kinds of data processing required for applications to both scientific research and e-commerce go well beyond what can be accomplished using only database query languages. Data must flow through a complex pipeline, being massaged and combined with other data in a great variety of ways; such processes are called **workflows**. In the case of scientific workflows, more than mere data translation or even complex data massaging are needed, including the use of powerful statistics packages, integration with complex scientific models, and display of selected data using visualization packages. Web-based business workflows have similar complexity, although the components are different. A number of languages have been developed to describe workflow components and processes, and of course there are also many *ad hoc* solutions. In general, it is a major task to construct a workflow for a specific project using these technologies, and therefore it is a major open problem to develop more flexible and user-friendly approaches. Service integration is a generalization of data integration, and the problems described above also arise in this new and more complex setting.

3 Abstract Schema Morphisms and Schema Matching

We assume familiarity with many-sorted algebra, e.g., [12, 14]. A **signature** is a set S of **sorts** and a set Σ of operation symbols with their **ranks**, each a string of input sorts and a single output sort. The **term algebra** T_Σ contains all well-formed Σ -terms, indexed by their sort. Given a **specification** (Σ, E) where E is a set of Σ -equations, the quotient T_Σ/E by the ground instances of equations in E is **initial**, in that it has a unique Σ -homomorphism to any other (Σ, E) -algebra.

Schemas describe those aspects of a database invariant under transactions, including both structural and integrity constraints. Let D be a fixed **data algebra**, of basic data elements and operations upon them, having one sort for each type of element, and including all elements as constants in its signature. Let T^D be the set of types of D , called **basic types**; then D is a T^D -sorted algebra. Typical elements of T^D are `Bool`, `Int`, `Char`, and `String` (of characters). Elements of D serve as data values in schema instances, i.e., databases or “documents.” We assume D has whatever constants, functions and relations are needed. Let P_i be a collection of algebraic specifications for **polymorphic types**, such as sets, bags, lists, or pairs; each P_i includes not only the constructors for that type, called **collectors**, but also **selectors** (which are inverses to constructors) and any needed predicates, given as Boolean valued functions. For each P_i , let c_i be an operation on types (not on data) with the same rank as the main constructor in P_i ; examples are unary `SetOf` and `ListOf`, and binary product \times . The constructors in P_i structure databases, for example, a relation as a set of records, whereas the c_i are constructors for type expressions.

We assume a set N of **names** for database elements, given as strings, i.e., elements of D_{String} . The **type signature** has one sort, plus the operations c_i , with constants including N and the sorts T^D of D . Call the term algebra of this signature the **type algebra**, denoted $T(N)$; its elements, called **type expressions**¹, are fundamental for abstract schemas (see Definition 6 below).

Example 1: Relational Databases. Let the name of the database itself be B , the names for its relations be R_1, \dots, R_n , and the names for the fields of each R_i be F_{ij} . Also let P_1 be a theory of finite sets with $c_1 = \text{SetOf}$, and let P_k for $k > 1$ define k -tuples, so that c_2 is the binary infix product type constructor \times , c_3 is a constructor for types of 3-tuples, etc.² Then

$$\mathcal{S}(B) = R_1 \times \dots \times R_n \quad \mathcal{S}(R_i) = \text{SetOf}(F_{i1} \times \dots \times F_{iK_i}) \quad \mathcal{S}(F_{ij}) \in T^D$$

The first says a relational database has relations R_i , the second says each R_i is a set of records of type $F_{i1} \times \dots \times F_{iK_i}$ having K_i fields F_{ij} which the third says all have values of a basic type. (If preferred, `BagOf` could be used instead of `SetOf`.) Any particular relational schema makes particular choices for these parameters, as illustrated below. \square

Example 2: A Relational Student Database. There are three relations, and

¹It seems not to be as widely known as it could be that it is often useful to regard type expressions as an initial algebra, and that initial algebras also elegantly capture the notion of abstract syntax; see [14].

²A more sophisticated approach uses only the binary pair constructor with an associative law, so that the n -tuple constructor is built with $n - 1$ pair constructors.

$S(B) = \text{Student} \times \text{Enrolled} \times \text{Course}$
 $S(\text{Student}) = \text{SetOf}(\text{StudentID} \times \text{Address} \times \text{Major} \times \text{GPA})$
 $S(\text{Enrolled}) = \text{SetOf}(\text{StudentID} \times \text{CourseId})$
 $S(\text{Course}) = \text{SetOf}(\text{CourseId} \times \text{Synopsis})$
 $S(\text{StudentID}) = \text{Int}$

We omit the remaining basic types for fields. The type algebra contains terms such as $\text{SetOf}(\text{CourseId} \times \text{Synopsis})$ and $\text{Student} \times \text{Enrolled} \times \text{Course}$, which uses the 3-tuple type constructor. \square

The **name graph** $G(S)$ of a partial function $S: N \rightarrow T(N)$ with a given top name B has B as its root node, and if t is a node of $G(S)$ and if a name n appears in $S(t)$, then $t.n$ is also a node of $G(S)$, and there is an edge from t to $t.n$ in $G(S)$. Then S is **acyclic** if its graph $G(S)$ is acyclic; this condition is useful for XML and similar structures, but not websites. Also, name $n \in N$ is **reachable** if it appears in $G(S)$, and S is **reachable** if every name in N is reachable; let N_* be the set of reachable names.

Definition 1. Given P , an **abstract schema** is a partial function $S: N \rightarrow T(N)$ that is acyclic with respect to a designated **top** name $B \in N$, and n' is a **match to** n if n' occurs in $\mathcal{M}(n)$. \square

Having explicit element names and treating tags and attributes the same way allows an element to match a tag or attribute in another schema.

Example 3: XML Schemas. The XML Schema spec is very complex (over 300 printed pages) and ugly, so we treat a simplification, which seems to agree with what Wadler calls “the essence of XML” [19], essentially an abstract schema with only one collector, ListOf , due to the inherent ordering in XML syntax. Here is the abstract schemas for a simple XML Schema for books, in which B is Bib , and

$S(\text{Bib}) = \text{ListOf}(\text{Book})$ $S(\text{Book}) = \text{title} \times \text{year} \times \text{Author}$ $S(\text{Author}) = \text{ListOf}(\text{author})$

If the specification for lists includes the empty list, then the constraint that Author lists are non-empty requires equation, for which see Example 7 below. \square

To define the databases that conform to a given abstract schema, we need a suitable signature; it is much larger than the signature of the type algebra.

Definition 2. For $S: N \rightarrow T(N)$ an abstract schema, its **explicit type algebra** $T^\#(N_*)$ is built like its type algebra, but with N replaced by N_* and with a new unary type constructor $\#n$ added for each $n \in N_*$. Let E denote the set of **type equations** of S , which are $n = \#n(S(n))$ for each $n \in N_*$. Then the **signature** $\Sigma(S)$ of S has the initial algebra $T^\#(N_*)/E$ as its sorts, plus the signature of each P_i instantiated with each element of $T^\#(N_*)/E$. \square

The $\#$ operations make types explicit, so that, e.g., $\#\text{title}(\text{String})$ and $\#\text{author}(\text{String})$ are different; also, paths can be traced by following $\#$ operations in parse trees of explicit type expressions.

Example 4: Bibliographic Schema Signature. The sorts are type expressions like $\text{ListOf}(\text{Book})$, $\text{Book}(\text{title} \times \text{year} \times \text{Author})$, and $\text{ListOf}(\text{title} \times \text{year} \times \text{ListOf}(\text{author}))$, which satisfy the type equations, so that Bib is equal $\#\text{Bib}(\text{ListOf}(\text{Book}))$ and to

$\#\text{Bib}(\text{ListOf}(\#\text{Book}(\#\text{title}(\text{String}) \times \#\text{year}(\text{Int}) \times \text{ListOf}(\#\text{Author}(\text{ListOf}(\#\text{author}(\text{String})))))))$

Unused names like GenomeRef do not appear in $\Sigma(S)$ because $\Sigma(S)$ is based on N_* not N . \square

Definition 3. The **specification** of an abstract schema $S: N \rightarrow T(N)$ has $\Sigma(S)$ as its signature, and has as its equations those of the P_i instantiated with all explicit type expressions, plus a spec for D . \square It is also convenient to remove “unreachable” types, not in any type expression equal to the top sort.

Example 5: Bibliographic Schema Specification. If the polymorphic specification P_1 for lists has binary constructor cons and selectors head and tail , then for any explicit type expression t , the equation $\text{head}(\text{cons}(a, L)) = a$ is included, for a a variable of sort t and L a variable of sort $\text{ListOf}(t)$. Note that this gives an infinite set of equations; another such set has equations $\text{tail}(\text{cons}(a, L)) = L$; all are in E . \square

Definition 4. Let I_S be an initial algebra of the specification of S . Then a **database** of S is an element of the carrier of the top sort of I_S . \square

Note that such elements can be described by terms, but the implementation in I_S may be quite different.

Example 6: Student Database. The explicit type expression for the top sort of Example 2 is

$\#\text{B}(\text{SetOf}(\#\text{Student}(\#\text{StudentID}(\text{Int}) \times \#\text{Address}(\text{String}) \times \#\text{GPA}(\text{Real}))) \times \text{SetOf}(\#\text{Enrolled}(\#\text{StudentID}(\text{Int}) \times \#\text{CourseId}(\text{String}))) \times \text{SetOf}(\#\text{CourseId}(\text{String}) \times \#\text{Synopsis}(\text{String}))))$

Then a typical small student database for the schema of Example 2 is:

$$\{ \langle 215, \text{Muir } 129, \text{Math}, 3.2 \rangle, \langle 329, \text{Revelle } 774, \text{CS}, 3.8 \rangle \}, \{ \langle 215, 130 \rangle, \langle 215, 220 \rangle, \langle 329, 130 \rangle, \langle 329, 87 \rangle \}, \\ \{ \langle 130, \dots \rangle, \langle 87, \dots \rangle, \langle 220, \dots \rangle \}$$

where ... indicates omitted text. \square

Integrity constraints and schema morphisms need notation for selectors in the P_i . If there is a standard notation, we will use it, e.g., `head` and `tail` for lists; for the product type constructor \times , we use its argument type names prefixed by $\&$. Also, for $n \in N$, let $\#\#n$ denote the selector from B to n , using iteration of conjunction over elements in collections (called “mapcar” in LISP).

Definition 5. An **integrity constraint** for schema S is an equation of the form $(\forall d: B) t(d) = \text{true}$, using operations in the signature of S . A **constrained abstract schema** is (S, C) where C is a set of constraints for S . A database m of S **satisfies** a constraint if $t(m) = \text{true}$, and **satisfies** (S, C) if it satisfies each constraint in C , in which case we write $m \models (S, C)$. \square

Example 7: Integrity Constraints. The following are typical integrity constraints for Example 3:

$$(\forall Y: \text{year}) Y \leq 2004 \quad (\forall A: \text{Author}) |A| > 0$$

where $| _ |$ is a length function assumed to be in the list specification (these do not look like equations, but they are with “ \leq ” and “ $>$ ” as Boolean valued functions followed by an implicit “ $= \text{true}$ ”.) Even so, these do not have the form required by Definition 5. However, they can be put in that form as follows:

$$(\forall d: B) \#\#\text{year}(d) \leq 2004 \quad (\forall d: B) |\#\#\text{Author}(d)| > 0$$

Typical constraints with more than one variable translate to forms with more than one $\#\#$. \square

The example below has a mapping with both **semantic functions**, which manipulate data values (e.g., for converting meters to feet), and **conditions**, which restrict the application of mapping formulae.

Example 8: Schema Mappings. Let the abstract schema S_1 be like S of Example 3 except for adding the following type definitions:

$$S_1(\text{author}) = \text{fname} \times \text{lname} \quad S_1(\text{fname}) = \text{NString} \quad S_1(\text{lname}) = \text{NString}$$

where NString is a basic type having no space character, and where fname and lname are elements for the first and last names of authors. Let S_2 be a second abstract schema built on S of Example 3 by adding the following type definition and constraint:

$$S_2(\text{author}) = \text{ListOf}(\text{NString}) \quad (\forall A: \text{author}) 0 < |A| < 3$$

Now S_1 databases map to S_2 databases by

$$\mathcal{M}_{\text{author}}(A) = \&\text{fname}(A) \bullet \&\text{lname}(A)$$

where author is the type name from S_2 , A is a variable of sort author from S_1 , $\&\text{fname}$ and $\&\text{lname}$ are selector functions for the author product type in S_1 , and \bullet is the append operation on lists.

The converse mapping for these schemas involves both conditions and semantic functions:

$$\mathcal{M}'_{\text{lname}}(A) = A \quad \text{if } |A| = 1 \quad \mathcal{M}'_{\text{lname}}(A) = \text{tail}(A) \quad \text{if } |A| = 2 \\ \mathcal{M}'_{\text{fname}}(A) = \perp \quad \text{if } |A| = 1 \quad \mathcal{M}'_{\text{fname}}(A) = \text{head}(A) \quad \text{if } |A| = 2$$

where A is a variable of type author from S_1 , and where \perp is a null value. (Formally, each pair of equations should be one equation with a polymorphic `if_then_else_`.) \square

There is an important duality, under which \mathcal{M} maps databases one way, but maps queries the opposite way (although this paper does not treat queries). For N a name set and $n \in N$, let $\%n$ be a new variable symbol of sort n , and let $\%N = \{\%n \mid n \in N\}$. We are now ready for the main concept:

Definition 6. An **abstract schema morphism** from $S: N \rightarrow T(N)$ to $S': N' \rightarrow T(N')$ is a partial function $\mathcal{M}: N'_* \rightarrow I_S(\%N)$, which is I_S with $\%N$ adjoined as new constant symbols. Let $\mathcal{M}(m)$ denote the mapping of a S model m by \mathcal{M} . If (S, C) and (S', C') are constrained schemas, then a morphism \mathcal{M} is a **constrained abstract schema morphism** if $m \models (S, C)$ implies $\mathcal{M}(m) \models (S', C')$. \square

Example 8 cont. The schema morphism from S to S' in Example 8 is obtained from the formulas for \mathcal{M} in that example, replacing A by $\%\text{author}$, and assuming that any name not explicitly mentioned is assigned the “obvious” mapping, which is the identity function for leaf nodes, e.g., $\mathcal{M}_{\text{year}}(Y) = Y$ for Y the variable $\%\text{year}$ of sort year from S_1 , is the tupling function for the product constructor, and for the list constructor, is iteration (“mapcar”) of a given function over list elements. \square

Proposition 1. Abstract schemas and schema morphisms form a category, with the identity morphism on S given by $1_S(v) = v$, and with composition of morphisms $\mathcal{M}: N'_* \rightarrow I_S(\%N)$ with $\mathcal{M}': N''_* \rightarrow I_S(\%N')$ defined by (for the cogniscenti, this is a Kleisli composition)

$$\mathcal{M}'; \mathcal{M}(n'') = \mathcal{M}'(n'')[\%n' \leftarrow \mathcal{M}(n')]_{n' \in N'_i}$$

which is the result of substituting $\mathcal{M}(\%n')$ for each occurrence of n' in $\mathcal{M}'(\%n)$. Constrained schemas and their morphisms also form a category. \square

This result makes available many powerful concepts and results from category theory, including a nice notion of equivalence via isomorphism, the correct notions of product, sum, and more generally, limit and colimit, for abstract schemas. All concepts apply to schemas of different kinds, and colimits give an elegant and extremely general notion of heterogeneous schema integration. See [1] for database motivation for the following, and Section 4 for some definitions:

Proposition 2. Abstract schemas as signatures (but with morphisms in the opposite direction), with integrity constraints as sentences, and with databases as models, form an institution. Moreover, particular kinds of schemas (XML, relational, etc.) form particular sub-institutions. \square

3.1 A Schema Mapping Tool

Our laboratory has designed and built a tool called SCIA which supports integration and transformation of databases having schemas in DTD and XML Schema format [15, 17, 20]. Since fully automatic schema mapping generation is infeasible, this tool attempts to minimize total user effort by identifying the critical decision points, where user input can yield the largest reduction of future matching effort. A **critical point** is where a core context has either no good matches, or else has more than one good 1-to-1 match, where **core contexts** are the most important contextualizing elements for tags within their subtrees. Core context elements typically have a large subtree, and can be found by heuristics and/or user input. In interactive mode, the tool solicits user input at critical points, and then iterates until both user and tool are satisfied; in automatic mode, it does just one pass using default strategies. Each pass has four steps: linguistic and data type matching; structural matching; context check; and combining match results. Other tools only try to find the easiest 1-to-1 matches, leaving all other difficult matches for the user to do by hand [18]; semantic functions, and conditions are not treated at all, or are left for a different tool for view generation, whereas our tool integrates these functions. A major finding is that this approach can significantly reduce total user effort.

4 Ontology Integration

An ontology is just a theory over a logic, i.e., a set of sentences in that logic. Using ontologies to integrate data raises issues analogous to those discussed in Section 2.1: Mappings of ontologies over a single logic are well enough understood, so that cocones and colimits of ontologies can be used (as in [1, 3]), but to integrate ontologies over different logics, the notion of logic must be formalized, along with morphisms of theories over different logics, for which morphisms of logics will also be needed. Such issues can be addressed using **institutions** [10], which axiomatize the notion of logical system based on Tarski's idea that the *satisfaction* of a sentence by a model is fundamental. However, we need to parameterize sentences and models over signatures, rather than just assume a single fixed signature, as Tarski did. Institutions have been successfully applied to give semantics for powerful module systems [13], and multi-logic specification languages [8], databases [1], behavioral types and semantics for the object paradigm [11], as well as to generalize many results in classical model theory, such as Craig interpolation [7].

An **institution** consists of an abstract category *Sign* of signatures, a functor *Sen*: *Sign* \rightarrow *Set* for sentences, a functor *Mod*: *Sign*^{op} \rightarrow *Cat* for models, and a satisfaction relation \models_{Σ} between models and sentences such that for every signature morphism $f: \Sigma \rightarrow \Sigma'$, we have $f(M) \models_{\Sigma'} e$ iff $M \models_{\Sigma} f(e)$. A **theory** over an institution \mathcal{I} is a pair (Σ, E) where E is a set of Σ -sentences. A Σ -model M **satisfies** (Σ, E) iff $M \models_{\Sigma} e$ for all $e \in E$. The **model class** of a theory, $(\Sigma, E)^{\bullet}$, is the class of all models that satisfy the theory, and the **theory** \mathcal{M}^{\bullet} of a class \mathcal{M} of models is the class of all sentences that are satisfied by all models in \mathcal{M} . This situation is a Galois connection, which gives us notions of **closed** theory, i.e., such that $(\Sigma, E)^{\bullet\bullet} = (\Sigma, E)$, and closed model class. Then a **theory morphism** $(\Sigma, E) \rightarrow (\Sigma', E')$ over \mathcal{I} is a signature morphism $f: \Sigma \rightarrow \Sigma'$ such that $f(E) \subseteq E'^{\bullet\bullet}$, and theories with these morphisms form a category denoted $Th(\mathcal{I})$. This category has whatever colimits *Sign* has [10].

An **institution morphism** from \mathcal{I} to \mathcal{I}' consists of a functor $\Phi: \text{Sign} \rightarrow \text{Sign}'$ and two natural transformations, $\alpha_{\Sigma}: \text{Sen}'(\Phi(\Sigma)) \Rightarrow \text{Sen}(\Sigma)$ and $\beta_{\Sigma}: \text{Mod}(\Sigma) \Rightarrow \text{Mod}'(\Phi(\Sigma))$, such that $M \models_{\Sigma} \alpha_{\Sigma}(e')$ iff

$\beta_{\Sigma}(M) \models_{\Phi(\Sigma)} (e')$, for all signatures Σ in \mathcal{I} , Σ -models M in \mathcal{I} , and $\Phi(\Sigma)$ -sentences e' in \mathcal{I}' . Institutions with these morphisms form a category, which we denote Ins .

There is an alternative approach to formalizing ontologies, pursued for example, in [16], using *local logics* in the sense of Barwise and Seligman [2]; however, local logics are actually a special case of institutions, in which sets of “instances” are models, “types” are sentences, “classification” is satisfaction, the “consequence” relation give morphisms of sentences (as in a more general formulation in [10], but not in the definition above), and the category of signatures has just one object and one morphism. (Sentences in information flow theories are not just types, they are sequents of types.)

The logical heterogeneity of ontology languages creates a need to deal with multiple institutions at once. Let \mathcal{O} be some (finite) subcategory of institutions. Then $\mathit{GTh}(\mathcal{O})$ has objects (\mathcal{I}, Σ, E) where \mathcal{I} is in \mathcal{O} and (Σ, E) is a theory of \mathcal{I} . A **morphism** in $\mathit{GTh}(\mathcal{O})$ from (\mathcal{I}, Σ, E) to $(\mathcal{I}', \Sigma', E')$ consists of an institution morphism $(\Phi, \alpha, \beta): \mathcal{I} \rightarrow \mathcal{I}'$ and a signature morphism $f: \Sigma' \rightarrow \Phi(\Sigma)$ such that $E \subseteq \alpha_{\Sigma}(f(E'))^{\bullet\bullet}$. There is an “obvious” composition that makes $\mathit{GTh}(\mathcal{O})$ into a category; in fact, it is the Grothendieck category of the theory functor $Th: \mathcal{O} \rightarrow \mathit{Cat}$, and it is also the theory category of the Grothendieck institution [8] of \mathcal{O} viewed as a diagram, i.e., a (contravariant) functor from a small category to the category of institutions. The Grothendieck institution [8] is a remarkable construction of a single institution from an indexed family of institutions; its category of signatures is the Grothendieck category of the indexed category of signatures of the institutions involved.

It is known that if \mathcal{O} satisfies some reasonable conditions, then $\mathit{GTh}(\mathcal{O})$ is cocomplete. An immediate benefit of this is a powerful “module system” for combining ontologies with heterogeneous logics. This arises from the fact that any cocomplete institution supports a rich collection of constructions on its theories, including instantiation of parameterized theories, sums of theories, and more; such features are important for structuring large and/or complex theories to better support reuse and reasoning. We thus get a powerful method for structuring ontologies into modules, including inheritance and sums of modules, shared submodules, and modules parameterized by other modules, in the style made popular by ML, but originating in the Clear language [6], and further developed under the name **parameterized programming** [9]. An elegant categorical semantics for this is given in [13]. Another benefit of the Grothendieck construction is its ability to lift Craig interpolation from the individual institutions to the whole [7]. This theory will be important when we extend our tool to take advantage of ontologies, providing a sound basis for its design, and an elegant semantics for its operation.

5 Conclusions and Future Work

We plan to extend our data integration tool so that it can handle relational schemas, spreadsheets, etc. This should be relatively straightforward, since it only requires writing a pre-processor to convert abstract schemas into the internal form of the tool (which consists of a tree and a graph in RDF notation).

A next step is to extend the tool to make use of ontologies. A significant issue for this task is connecting ontologies to abstract schemas; a good discussion of the setting for this problem is given in [5]. One natural approach is to translate ontologies into abstract schemas. Some information will be lost, but not as much as might be feared; for example, a logical implication can be translated into an inclusion relation on sorts, as in order sorted algebra [12]. The full generality of institutions is too great for this, so we restrict to description logics or a similar class, the theories of which can be encoded as abstract schemas, so that the mapping generation tool can do the translation. It would be interesting to abstractly characterize the institutions for which this works.

Some more theoretical issues also need further investigation, especially the properties of various categories discussed above, including abstract schemas, constrained abstract schemas, and Grothendieck categories of ontologies over heterogeneous logics. Relationships with approaches based on local logics should also be further explored.

References

- [1] Suad Alagic and Philip Bernstein. A model theory for generic model management. In Giorgio Ghelli and Gösta Grahne, editors, *Proc. Database Programming Languages 2001*, pages 228–246. Springer, 2002.

- [2] Jon Barwise and Jerry Seligman. *Information Flow: Logic of Distributed Systems*. Cambridge, 1997. Tracts in Theoretical Computer Science 44.
- [3] Trevor Bench-Capon and Grant Malcolm. Formalising ontologies and their relations. In *Proceedings of the 16th International Conference on Database and Expert Systems Applications (DEXA '99)*, pages 250–259. Springer, 1999. Lecture Notes in Computer Science, volume 1677.
- [4] Philip Bernstein. Applying model management to classical meta data problems. In *Proc. Conf. on Innovative Database Research*, pages 209–220, 2003.
- [5] Shawn Bowers and Bertram Ludäscher. An ontology-driven framework for data transformation in scientific workflows. In *Data Integration in the Life Sciences*. Springer, 2004.
- [6] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.
- [7] Răzvan Diaconescu. Interpolation in grothendieck institutions. *Theoretical Computer Science*, 311:439–461, 2004.
- [8] Răzvan Diaconescu. Grothendieck institutions. *Applied Categorical Structures*, 10:383–402, 2002.
- [9] Joseph Goguen. Principles of parameterized programming. In Ted Biggerstaff and Alan Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.
- [10] Joseph Goguen and Rod Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, January 1992.
- [11] Joseph Goguen and Răzvan Diaconescu. Towards an algebraic semantics for the object paradigm. In Hartmut Ehrig and Fernando Orejas, editors, *Proceedings, Tenth Workshop on Abstract Data Types*, pages 1–29. Springer, 1994. Lecture Notes in Computer Science, Volume 785.
- [12] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs*. MIT, 1996.
- [13] Joseph Goguen and Grigore Roşu. Composition of hidden information modules over inclusive institutions. In *From Object-Oriented to Formal Methods: Essays in Honor of Johan-Ole Dahl*. Springer, to appear 2003.
- [14] Joseph Goguen, James Thatcher, Eric Wagner, and Jesse Wright. Initial algebra semantics and continuous algebras. *Journal of the Association for Computing Machinery*, 24(1):68–95, January 1977.
- [15] Joseph Goguen, Guilian Wang, Young-Kwang Nam, and Kai Lin. Abstract schema morphisms and schema mapping generation. Technical report, Dept. Computer Science and Engineering, UCSD, 2004. Submitted for publication.
- [16] Yannis Kalfoglou and Marco Schorlemmer. Information-flow-based ontology mapping. In Robert Meersman and Zahir Tari, editors, *Proc. Intl. Conf. on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems*, volume 2519 of *Lecture Notes in Computer Science*, pages 1132–1151. Springer, 2002.
- [17] Young-Kwang Nam, Joseph Goguen, and Guilian Wang. A metadata integration assistant generator for heterogeneous distributed databases. In Robert Meersman and Zahir Tari, editors, *Proc. Intl. Conf. on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems*, volume 2519 of *Lecture Notes in Computer Science*, pages 1332–1344. Springer, 2002.
- [18] Erhard Rahm and Philip Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.
- [19] Jérôme Siméon and Philip Wadler. The essence of XML. In *Proc. Principles of Programming Languages*, pages 1–13. ACM, 2003.
- [20] Guilian Wang, Joseph Goguen, Young-Kwang Nam, and Kai Lin. Critical points for interactive schema matching. In Jeffrey Xu Yu, Xuemin Lin, Hongjun Lu, and YanChun Zhang, editors, *Advanced Web Technologies and Applications*, pages 654–664. Springer, 2004.