A Model Theory for Generic Schema Management

Suad Alagić and Philip A. Bernstein
Microsoft Research
One Microsoft Way
Redmond, WA 98052-6399
alagic@cs.twsu.edu, philbe@microsoft.com

Abstract

The core of a model theory for generic schema management is developed. This theory has two distinctive features: it applies to a variety of categories of schemas, and it applies to transformations of both the schema structure and its integrity constraints. A subtle problem of schema integration is considered in its general form, not bound to any particular category of schemas. The proposed solution, as well as the overall theory, is based entirely on schema morphisms that carry both structural and semantic properties. Duality results that apply to the two levels (i.e., the schema and the data levels) are established. These results lead to the main contribution of this paper: a formal schema and data management framework for generic schema management. Implications of this theory are established that apply to integrity problems in schema integration. The theory is illustrated by a particular category of schemas with object-oriented features along with typical database integrity constraints.

1 Introduction

This paper presents the core results of a model theory for generic schema management, by which we mean schema and database transformation capabilities that are independent of a particular data model. Such transformations require major database programming tasks, such as integrating source schemas when building a data warehouse or integrating different user views into an overall database schema. In spite of nontrivial typing issues created by such transformations, database programming and other relevant paradigms have been primarily suited to dealing with structural aspects of those transformations. A major challenge is in properly addressing semantics: the integrity constraints associated with database schemas.

A second major challenge is in developing such a model theory that is applicable to a variety of data models, such as the relational, object-oriented, and XML models [23]. This is challenging because schemas and their underlying databases are very different in these three major categories of models as are languages and their underlying logics used for expressing the integrity constraints.

On the pragmatic side, generic schema management operations and tools have been considered in [4], and more specific system implications in [5, 20]. These papers argue that many difficult and expensive database programming problems involve the manipulation of mappings between schemas. Examples are populating a data warehouse from data sources, exposing a set of data sources as an integrated schema, generating a web site wrapper, generating an object-oriented wrapper for relational data, and mapping an XML schema to a relational schema. Despite many commonalities of these schema management problems, tools and languages are typically engineered for only one problem area. A more attractive approach would be to build generic schema management tools and languages that would apply to all of these problems, with only some customization required for the data model and problem at hand. The formal open problem is to find a suitable model theory that would be able to handle this generality.

Our proposed model theory has a categorical flavor, manifested in the use of arrows that appear at two levels. At the meta level the arrows represent schema transformations. For example, an arrow could map a data source schema to a data warehouse schema, or two arrows could map each of two data source schemas into a mediated schema. These transformations are defined as schema morphisms that map the structural properties and integrity constraints of a schema. At the data level the arrows are data transformations specified as database morphisms. Database morphisms map the actual data sets in a manner that is compatible with the operations available on those sets and that preserves the schema's integrity constraints.

There are several implications of this arrow-theoretic approach. The first is that it leads to a very general view of the schema integration problem expressed entirely in terms of arrows. The generality is accomplished by using a particular categorical construction which applies to a variety of categories of schemas [10, 12, 17]. The specific nature of arrows used in this construction is determined by the category

of schemas that defines the kind of structural and integrity transformations that the arrows actually represent.

The second implication is in the formal results that relate properties of arrows at the meta level and the data level. These results reveal a subtle duality manifested in the reversal of the corresponding arrows between the two levels. They also provide guidance for how to define schema transformations appropriately, in particular, so that the arrows preserve integrity constraints.

The third, most important implication is a formal framework for generic schema and data management which applies to a variety of data models. This is really the main contribution of this paper. This formal framework includes and relates the two levels (schema and data transformations) and captures both structural properties and integrity constraints. A key component of this theory is a strong integrity requirement on the permissible structural schema transformations so that they preserve the integrity constraints. This model theory relies on earlier results on a general model theory for a variety of programming paradigms and of associated logic paradigms [9, 10].

We apply the proposed formal theory to two situations. First, we apply it to the schema integration problem where we prove results that are generic across data models and include a proper treatment of the integrity constraints. Second, we use it as a pattern for defining a data model, one based on abstract data types, which includes object-oriented features and typical database integrity constraints.

The paper is organized as follows. Section 2 starts with basic definitions, followed by a running example of a particular category of object-oriented schemas. The basic categorical definitions are given in Section 3, particularly the use of morphisms to represent schema mappings. Sections 4 through 6 are the core of the paper. Section 4 shows how to express the schema integration problem using morphisms. Section 5 introduces the notion of a generic schema transformation framework, specifies the condition for such a framework to preserve database integrity and proves the implications on schema integration. Section 6 shows how the proposed model theory applies to the category of schemas of the running example. Section 7 discusses related work and concludes.

2 Schemas

A database schema consists of two components: a schema signature and the associated integrity constraints. A schema signature specifies structural and operational features of a database schema. Its typical components are signatures for data types and their operations and signatures for *database sets* (relations, collections, etc.), illustrated in Example 1. Note that type signatures for collections determine the signatures for operations on collections. Signatures for integrity constraints are logical expressions whose form is determined by the choice of a particular logic and syntax of the constraint language. A feature of the definitions below is that they are sufficiently general to apply to a variety of schema signatures and associated constraint languages.

Definition 1 *(Schemas) A database schema Sch is a pair $Sch = (Sig, E)$ where Sig is a schema signature and E is a set of integrity constraints expressed as sentences (formulae of a chosen logic with all variables quantified).*

This paper uses examples based on a category of schemas of an object-oriented style. Schemas of this category have user-defined abstract data types specified as Java interfaces. A schema signature consists of the specification of those types and of collection objects which represent the actual database. The integrity constraints are specified in Horn clause logic. We use only the most typical database constraints: those expressing key dependencies and referential integrity constraints.

Example 1 *(Sample schema)*

```
schema Publishers {
interface Publisher
{ String publisherId();
  String name();
  String location();
  Set<Publication> publications();
}
interface Publication
{ String publicationId();
  String title();
  int    year();
  Publisher   publisher();
```

```
  Set<String> keywords();
}
Collection<Publisher>   dbPublishers;
Collection<Publication> dbPubs;
Constraints:
Publication X,Y;  Publisher Z,W;
dbPubs.member(X)       :-  dbPublishers.member(Z), Z.publications.member(X);
dbPublishers.member(Z) :-  dbPubs.member(X), X.publisher.equals(Z);
Z.equals(W) :- dbPublishers.member(Z), dbPublishers.member(W),
               Z.publisherId().equals(W.publisherId());
X.equals(Y) :- dbPubs.member(X), dbPubs.member(Y), X.publicationId().equals(Y.publicationId());
}
```

As Java interfaces lack any kind of general-purpose constraint specifications, such constraints are omitted from our examples. However, this omission is by no means a limitation of our approach (see [1, 2]). Note that `equals` is a method of the Java root class `Object` which is intended to be overridden to provide a meaning of equality specific to a particular data type. If this is the standard notion of equality then the logic paradigm becomes Horn clause logic with equality as in [11].

A database is a model for a schema. Given a schema signature Sig, the collection of all databases that *conform to* Sig (*implement* Sig) is denoted by $Db(Sig)$. A database d that implements Sig would thus have to implement the signatures of data types in Sig as sets and the operation signatures as functions. Signatures for database sets would also have to be interpreted as sets, bags etc.

If a database d that conforms to a schema signature Sig satisfies the integrity constraints E, we say that d is consistent with respect to the schema (Sig, E). This is expressed by the satisfaction relation, denoted $\models$, between databases and the sets of sentences (i.e., integrity constraints) that they satisfy.

Definition 2 *(Database consistency) A database d is consistent with respect to the schema (Sig, E) iff d belongs to $Db(Sig)$ and $d \models e$ for all $e \in E$.*

3 Schema Morphisms

In this approach the schema signatures for a given data model are required to constitute a category, denoted **Sig**. The same applies to the schemas. Schema transformations within a particular category of schemas are viewed as morphisms of that category. This approach allows us to talk about schemas without specifying their data model (i.e. category).

Formally, a *category* **C** consists of the following [17]:

- A collection of objects and a collection of arrows (morphisms).
- Each arrow has its domain object and its codomain object.
- If $f : X \to Y$ and $g : Y \to Z$ are arrows of **C**, then f and g are composed into an arrow $gf : X \to Z$.
- The composition of arrows is associative, i.e., if $f : X \to Y$, $g : Y \to Z$ and $h : Z \to W$ are arrows of **C**, then $h(gf) = (hg)f$.
- Each object X of **C** is equipped with the identity arrow 1_X with the property that for any arrow $f : X \to Y$, $1_Y f = f$ and $f1_X = f$.

The collection of all databases $Db(Sig)$ conforming to a given schema signature Sig is required to constitute a category, with database morphisms that satisfy the above categorical requirements.

Schema morphisms are defined as mappings of schema signatures that preserve the integrity constraints.

Definition 3 *(Schema morphisms) A morphism of schemas $Sch_1 = (Sig_1, E_1)$ and $Sch_2 = (Sig_2, E_2)$ consists of a morphism $\phi : Sig_1 \to Sig_2$ of schema signatures which extends to a mapping of constraints, such that for all databases d in $Db(Sig_2)$, if $d \models e_2$ for all $e_2 \in E_2$ then $d \models \phi(e_1)$ for all $e_1 \in E_1$.*

An equivalent condition is that $\phi(e_1)$ is in the closure of E_2 for all $e_1 \in E_1$. A particular case is $\phi(e_1) \in E_2$.

Definition 3 implies that schemas and their morphisms constitute a category, which we denote by **Sch**. The last condition in Definition 3 differentiates this work from many others. It requires that the integrity constraints of the source schema are transformed into constraints that are consistent with those of the target. This is expressed in a model-oriented fashion: a database that satisfies the constraints of the target schema also satisfies the transformed constraints of the source schema (as they appear in the target).

In Definition 1, distinguishing the notions of schema signature and schema (i.e., a schema signature plus its integrity constraints) leads to two notions of schema equivalence. The first one is just structural, based on schema signature morphisms. The second is semantic, requiring structural equivalence plus the semantic equivalence expressed in terms of integrity constraints.

Definition 4 *(Schema equivalence) Let Sch_1 and Sch_2 be two schemas*

- *Sch_1 and Sch_2 are structurally equivalent if there exists a pair of schema signature morphisms $f : Sig_1 \to Sig_2$ and $g : Sig_2 \to Sig_1$ such that $fg = 1_{Sig_2}$ and $gf = 1_{Sig_1}$.*
- *Sch_1 and Sch_2 are equivalent if the above condition is satisfied for schema morphisms f and g.*

The category of all schemas for a given data model is required to include the initial schema Sch_0 that contains basic features implicitly available in any other schema (for example, predefined standard types such as Boolean, integer and string, and their associated constraints). Sch_0 typically does not contain signatures for database sets but does contain type signatures for the required collection types. Any other schema Sch_X implicitly extends Sch_0 by a unique schema morphism $Sch_0 \to Sch_X$. The uniqueness requirement means that there is one standard way of incorporating Sch_0 into Sch_X which makes Sch_0 a subschema of Sch_X.

The notion of a subschema Sch_1 of Sch_2 may be expressed by the categorical requirement that a monic arrow (schema morphism) $m : Sch_1 \to Sch_2$ exists. This means that given schema morphisms $f : Sch_X \to Sch_1$ and $g : Sch_X \to Sch_1$, $mf = mg$ implies $f = g$ [17]. In familiar cases, monic arrows are injections.

4 Schema Integration

A particularly important problem in schema management is schema integration [6, 21, 22]. Here, we are given two (or more) schemas and are asked to produce an integrated schema that can represent the information content of the given schemas. In one version of the problem, which we treat here, the integrated schema is populated with data and the given schemas are defined as views of the integrated schema.

All published schema integration results we know of are tied to a particular data model (i.e., to a particular category of schemas). By contrast, our approach applies to different categories of schemas. Moreover, it addresses the subtle problem of merging the integrity constraints of two schemas. These two distinctive features are accomplished by expressing the idea of schema integration in terms of morphisms of schemas. This way both structural and semantic conditions are taken into account.

Definition 5 *(Schema integration) An integration Sch_{12} of schemas Sch_1 and Sch_2 is defined by the following commutative diagram of schema morphisms:*

$$\begin{array}{ccc} Sch_m & \xrightarrow{\phi_1} & Sch_1 \\ \phi_2\downarrow & & p\downarrow \\ Sch_2 & \xrightarrow{q} & Sch_{12} \end{array}$$

Two schemas are integrated over their matching part Sch_m. Two schemas always have a matching part: the initial schema Sch_0. The matching part Sch_m is typically a subschema of both Sch_1 and Sch_2. The commutativity of the above diagram asserts that the two composite schema morphisms $p\phi_1$ and $q\phi_2$ are identical. This means that Sch_m appears the same in Sch_{12} whichever path (via Sch_1 or Sch_2) is taken.

Example 2 *(Schema integration) We show the integration Sch_{12} of Sch_1 (in Example 1) with another schema, Sch_2, such that Sch_1 and Sch_2 can be defined as views of Sch_{12}. First, we define Sch_2 as follows:*

```
schema Authors {
interface Author
{ String authorId();
  String name();
  Date   dateOfBirth();
  Collection<Book> books();
 }
interface Book
{ String publicationID();
```

```
  String title();
  int    year();
  String publisher();
  Set<String> keywords();
  float  price();
  Collection<Author> authors();
}
Collection<Author> dbAuthors;
Collection<Book>   dbBooks;

Constraints:
Author X,Y; Book Z,W;
dbBooks.member(Z)   :-  dbAuthors.member(X), X.books.member(Z);
dbAuthors.member(X) :-  dbBooks.member(Z),   Z.authors.member(X);
X.equals(Y)   :-  dbAuthors.member(X), dbAuthors.member(Y), X.authorId().equals(Y.authorId());
Z.equals(W)   :-  dbBooks.member(Z),   dbBooks.member(W),   Z.ISBN().equals(W.ISBN())
}
```

A schema Sch_{12} that integrates Sch_1 and Sch_2 is given below:

```
schema Publications {
import Publishers.Publisher;
import Publishers.Publication;
import Authors.Author;
interface Book extends Publication // integrates Book and Publication in Sch12
{ float price();
  Collection<Author> authors();
}
Collection<Publisher>   dbPublishers;
Collection<Publication> dbPubs;
Collection<Author> dbAuthors;
Collection<Book>   dbBooks;
Constraints:
import Publishers.Constraints;
import Authors.Constraints;
Publication X;
dbPubs.member(X) :- dbBooks.member((Book)X) // an additional constraint beyond Sch1 and Sch2
}
```

Among all integrations Sch_{12} of schemas S_1 and S_2, the schema-join of Sch_1 and Sch_2 denoted $Sch_1 * Sch_2$, if it exists, has a distinctive property: it represents the minimal integration of Sch_1 and Sch_2. This notion is specified below entirely in terms of schema morphisms by a categorical construction called a *pushout* [17].

Definition 6 *(Schema join) Schema-join $Sch_1 * Sch_2$ of schemas Sch_1 and Sch_2 is defined by the following commutative diagram of schema morphisms:*

$$\begin{array}{ccc} Sch_m & \xrightarrow{\phi_1} & Sch_1 \\ \phi_2\downarrow & & h\downarrow \\ Sch_2 & \xrightarrow{k} & Sch_1 * Sch_2 \end{array}$$

with the following property: Given any integration Sch_{12} of schemas Sch_1 and Sch_2 as defined in Definition 5 above, there exists a unique schema morphism $\phi : Sch_1 * Sch_2 \rightarrow Sch_{12}$ such that $\phi k = q$ and $\phi h = p$.

A specific construction of schema-join is presented in Section 6. The fact that schema integration and schema-join are expressed entirely in terms of arrows representing schema morphisms has two distinctive implications: (i) both structural and database integrity properties are integrated, and (ii) the notion of schema integration is data model independent. Specific notions of schema integration are obtained by choosing a particular category of schemas which includes its schema morphisms.

We believe that the notion of least upper bound in the schema lattice of [7] and the schema + operator of [19] can be modeled precisely as schema join. Our category-theoretic treatment generalizes [7] by considering the instance level, not just the schema level and generalizes both models by making it applicable to a variety of schema categories, including such features as methods and constraints.

5 Generic Schema Transformation Framework

The categorical approach developed so far leads to a very general notion of a data model. This model has two levels. The meta level consists of database schemas and their transformations expressed as schema morphisms. This transformation-based view is quite different from the standard notions of a data model. At the instance level these transformations operate on databases that conform to schemas at the meta level.

The notion of database integrity plays a fundamental role in this formal framework. The acceptable transformations at both levels are required to satisfy the integrity constraints. This integrity requirement is expressed as a condition that involves acceptable schema signature transformations and the satisfaction relation between databases and the integrity constraints.

The framework is proposed as a formal pattern for defining data models such that schema management and the associated database transformations have well-defined formal meanings. As such, it offers interesting observations on the relationships between the two levels which are sometimes quite different from the usual views of schema and data transformations. An example is the reversal of the directions of transformations at the two levels. Schema management operations such as schema integration have particularly desirable semantic properties when the conditions for this formal framework are satisfied.

The core of this model theory requires just one more categorical notion: a morphism of categories. Given categories $\mathbf{C}$ and $\mathbf{B}$, a *functor* $F : \mathbf{C} \to \mathbf{B}$ consists of two related functions with the following properties [17]:

- The object function which assigns to each object X of $\mathbf{C}$ an object FX of $\mathbf{B}$.
- The arrow function which assigns to each arrow $h : X \to Y$ of $\mathbf{C}$ an arrow $Fh : FX \to FY$ of $\mathbf{B}$.
- $F(1_X) = 1_{FX}$ and $F(gh) = F(g)F(h)$, the latter whenever the composite gh is defined.

Two functors play a crucial role in this theory. The first one is $Sen : \mathbf{Sign} \to \mathbf{Set}$ where $\mathbf{Set}$ denotes the category of sets. If Sig is a schema signature, then $Sen(Sig)$ is the set of all well-formed sentences (integrity constraints) over Sig. Many constraint languages, hence many Sen functors, are possible for a given data model (i.e., which implies a choice of $\mathbf{Sign}$). Sen is determined by the choice of logic that specifies the syntax of sentences of the constraint language. This syntax is defined starting with the features of a schema signature. The schema signature typically determines the terms of the constraint language, and the logic determines the formulae based on those terms. This is why Sen maps a schema signature into a set of sentences over that signature. Sen also maps a schema signature morphism to a function that transforms the sets of sentences.

The second functor is $Db : \mathbf{Sign} \to \mathbf{Cat}^{op}$. For each signature Sig, $Db(Sig)$ is the category of Sig databases, together with their morphisms. These database morphisms represent data transformations, which correspond to the schema transformations represented by arrows in $\mathbf{Sign}$. $\mathbf{Cat}$ denotes the category of categories. Objects of $\mathbf{Cat}$ are categories and arrows of $\mathbf{Cat}$ are functors. $\mathbf{Cat}^{op}$ differs from $\mathbf{Cat}$ only to the extent that the direction of its arrows is reversed. This reversal of the direction of arrows that happens going from schemas to their databases is one of the subtle and characteristic features of this model theory.

Recall that databases in the category $Db(Sig)$ are not necessarily consistent with respect to a set of integrity constraints E. The notion of database integrity is captured by the satisfaction relation $\models$ between databases in $Db(Sig)$ and sets of sentences over Sig.

We now have the machinery to define the notion of data model described in the beginning of this section. To emphasize the transformation-based view relative to classical data models, we give it a new name.

Definition 7 *(Schema transformation framework) A schema transformation framework consists of:*

- *A category of schema signatures* $\mathbf{Sign}$ *equipped with the initial object. This category consists of objects representing schema signatures together with their morphisms.*
- *A functor* $Sen : \mathbf{Sign} \to \mathbf{Set}$. $Sen(Sig)$ *is a set of sentences over the schema signature* Sig.
- *A functor* $Db : \mathbf{Sign} \to \mathbf{Cat}^{op}$. *For each signature* Sig, $Db(Sig)$ *is the category of* Sig *databases, together with their morphisms.*
- *For each signature* Sig, *a relation* $\models_{Sig} \subseteq |\, Db(Sig)\,| \times Sen(Sig)$ *called the satisfaction relation.* $|\, Db(Sig)\,|$ *denotes the set of objects of the category* $Db(Sig)$.
- *For each schema signature morphism* $\phi : Sig_A \to Sig_B$, *the following* Integrity Requirement *holds for each* Sig_B *database* d_B *and each sentence* $e \in Sen(Sig_A)$:

$$d_B \models_{Sig_B} Sen(\phi)(e) \text{ iff } Db(\phi)(d_B) \models_{Sig_A} e.$$

The above definition is based on [9]. Its relationships are represented by the following diagram:

$$\begin{array}{ccc} Db(Sig_A) & \stackrel{\models_{Sig_A}}{\longrightarrow} & Sen(Sig_A) \\ Db(\phi)\uparrow & & \downarrow Sen(\phi) \\ Db(Sig_B) & \stackrel{\models_{Sig_B}}{\longrightarrow} & Sen(Sig_B) \end{array}$$

Note the reversal of the direction of the arrow $Db(\phi)$ relative to $Sen(\phi)$. $Sen(\phi)$ maps each constraint in $Sen(Sig_A)$ to a constraint in $Sen(Sig_B)$. By contrast, $Db(\phi)$ maps each database in $Db(Sig_B)$ to a database in $Db(Sig_A)$. If we think of ϕ as a mapping from a logical database schema Sig_A into a physical schema Sig_B, then $Db(\phi)$ tells how to materialize a view in $Db(Sig_A)$ from a database in $Db(Sig_B)$.

To see why this reversal of arrows happens, consider an injective schema morphism $\phi : Sch_1 \to Sch_2$ which makes Sch_1 a subschema of Sch_2. Sch_2 just extends the signatures and the constraints of Sch_1 (we assume a monotonic logic such as Horn clause logic). A database that is consistent with respect to Sch_2 is also (by projection) consistent with Sch_1. The other way around does not hold.

The *Integrity Requirement* puts a very strong semantic restriction on the permissible schema signature transformations (schema morphisms). It only allows ones that preserve the validity of constraints. That is, suppose ϕ maps constraint e_A to e_B (more precisely, $e_B = (Db)(\phi)(e_A)$ for $e_A \in Sen(Sig_A)$). Then the *Integrity Requirement* says that e_B is valid in a database d_B iff e_A is valid in the Sig_A database that corresponds to d_B (i.e., in $Db(\phi)(d_B)$).

The *Integrity Requirement* applies directly to the problem of schema integration in Definition 6. It ensures that each valid database of the integrated schema corresponds to valid databases of the schemas to be integrated. This point is made precise in the following theorem.

Theorem 1 *(Materializing subschemas) Suppose Sch_1 and Sch_2 are schemas that are integrated into Sch_{12} relative to some schema transformation framework. Given a consistent database d in $Db(Sch_{12})$, it is possible to construct databases d_1 and d_2 that are consistent relative to schema Sch_1 and schema Sch_2 respectively.*

Proof We have schema morphisms $\phi_1 : Sch_1 \to Sch_{12}$ and $\phi_2 : Sch_2 \to Sch_{12}$ where $Sch_1 = (Sig_1, E_1)$ and $Sch_2 = (Sig_2, E_2)$. We construct database d_1 as $Db(\phi_1)(d)$ and database d_2 as $Db(\phi_2)(d)$.

Since d is a consistent database and ϕ_1 and ϕ_2 are schema morphisms, Definition 3 implies:

- $d \models_{Sig_{12}} (Sen)(\phi_1)(e)$ for all $e \in E_1$
- $d \models_{Sig_{12}} (Sen)(\phi_2)(e)$ for all $e \in E_2$.

We can now complete the proof by applying the Integrity Requirement to the above two lines, yielding:

- $Db(\phi_1)(d) \models_{Sig_1} e$ for all $e \in E_1$
- $Db(\phi_2)(d) \models_{Sig_2} e$ for all $e \in E_2$.

In essence, databases conforming to Sch_1 and Sch_2 are materialized views of Sch_{12}. Thus, the above theorem relates the consistency of databases conforming to the integrated schema to those of the materialized views. This is an unusual perspective in that views typically do not have integrity constraints.

Note that in order to make this proof possible both the reversal of arrows and the Integrity Requirement are essential.

Corollary 1 *(Schema-joins) Let Sch_{12} be an integration of schemas Sch_1 and Sch_2 relative to a schema transformation framework. Let d be a consistent database in the category $Db(Sig_{12})$. If $Sch_1 * Sch_2 = (Sig_{1*2}, E_{1*2})$ exists, then there is a canonical way to construct a consistent database d_* in the category $Db(Sig_{1*2})$.*

Proof By Definition 6, there is a unique $\phi : Sch_1 * Sch_2 \to Sch_{12}$. d_* is constructed as $Db(\phi)(d)$. Since d is consistent, by Theorem 1 so is d_*.

Corollary 1 says that for any consistent database of an integrated schema, there exists a corresponding (minimal) consistent database of the schema-join of the two schemas that were integrated. The canonical arrow $Db(\phi)$ is a recipe for constructing that (minimal) consistent database.

6 An Application of the Schema Transformation Framework

One role of the generic schema transformation framework is to serve as a formal generic pattern for defining new data models, with well defined meanings for schema and data transformations. This section gives a detailed description of how one constructs such a data model: a category of schemas and their databases for Examples 1 and 2, which we call **Objc** (OO with Constraints).

A schema of **Objc** consists of a collection of sorts (type names) some of which are predefined in the initial schema Sch_0 and which in particular must include the sort Boolean along with the standard axioms. The others are abstract data types defined in the schema itself, each of which is a set of method signatures specified as a Java interface. The is-a (inheritance) relationship thus amounts to the subset relation which agrees with the rules of the Java type system. A schema contains collection objects which are the actual database sets. To specify their type, a parametric $Collection < T >$ type is used and instantiated with the required element type selected among the abstract data types defined in the schema.

Definition 8 *(**Objc** schema signatures) A schema signature consists of:*

- *A finite set of sorts S, which includes Boolean.*
- *A finite set of interfaces A (abstract data types) such that $A \subseteq S$.*
- *An interface A_k is a set of method signatures of the form $C\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n)$ where $C \in S$ and $C_i \in S$ for $i = 1, 2, \ldots, n$. (m is the method name, C is the return type, and C_i is the type of parameter x_i.)*
- *If the interface A_k extends the interface A_l (inheritance) then $A_l \subseteq A_k$.*
- *A finite set of collection signatures $\{Collection < A_j >: X_j\}$ where $A_j \in A$ and $Collection < A_j > \in S$.*

To specify the type of data sets of a schema, parametric collection types are required. The issue of parametric types is a major one by itself. It will be elaborated only to a limited extent here by showing that a specific collection type (i.e., for a specific element type) is obtained from a parametric type by the same pushout construction [10] that we used for schema integration.

Definition 9 *(**Objc** collection types) An instantiated collection interface $Collection < A >$ is defined by the following pushout diagram*

$$\begin{array}{ccc} T & \longrightarrow & Collection < T > \\ \downarrow & & h\downarrow \\ A & \stackrel{k}{\longrightarrow} & Collection < A > \end{array}$$

In the above definition a parametric interface $Collection < T >$ is viewed as a morphism $T \rightarrow Collection < T >$. The substitution $T \rightarrow A$ and instantiation $Collection < T > \rightarrow Collection < A >$ are also morphisms. The morphism $A \rightarrow Collection < A >$ is obtained from $T \rightarrow Collection < T >$ and the substitution $T \rightarrow A$.

The above construction generalizes the usual views of instantiated parametric types. If a pushout is based on schema morphisms (which apply to both structural properties and integrity constraints), not just on signature morphisms, then the notion of instantiation of parametric types becomes semantic in nature.

Constraints are sentences expressed in Horn clause logic. Terms include method invocations and thus appear in the object-oriented form. Atoms are invocations of Boolean methods. This limited logic is sufficiently expressive to capture most typical database constraints, such as key and inclusion dependencies.

Definition 10 *(**Objc** constraints) For a given schema signature Sig with sorts S:*

- *A collection object of type $Collection < A_k >$ is a term of type $Collection < A_k >$.*
- *A variable X of type A_k where $A_k \in S$ is a term of type A_k.*
- *If a is a term of type A_k, $a_1, a_2, \ldots, a_n$ are terms of respective types $A_1, A_2, \ldots, A_n$, $C\ m(A_1, A_2, \ldots, A_n)$ is a method of the interface A_k of Sig, then $a.m(a_1, a_2, \ldots, a_n)$ is a term of type C.*
- *With the above definitions, an atom is of the form $a.m(a_1, a_2, \ldots, a_n)$ where m's result type is Boolean.*
- *A constraint is of the form $p \leftarrow p_1, p_2, \ldots, p_n$ where $p, p_1, p_2, \ldots, p_n$ are atoms.*

Permissible schema signature transformations of **Objc** are defined below as schema signature morphisms. Their core is a mapping of sorts, extended to signatures of methods and collection objects. This mapping is the identity on the inital schema's predefined sorts, so these sorts have the same interpretation in all **Objc** schemas. Note that $Collection < T >$ is not intended to have methods with arguments of type $Collection < T >$, so that $Collection < A_k >$ would be a structural subtype of $Collection < A_l >$ if A_k is a subtype of A_l.

Definition 11 *(***Objc** *schema signature morphisms) A morphism of schema signatures* $\phi : Sig_1 \to Sig_2$ *consists of the following:*

- *An injective mapping of sorts* $\phi : S_1 \to S_2$ *such that*
  - $\phi(s) = s$ *for* $s \in S_0$ *where* S_0 *are the sorts of the initial schema* Sch_0.
  - $\phi(Collection < A_i >) = Collection < \phi(A_i) >$
- ϕ *applies to interfaces as follows:* $\phi(C\ m(C_1,\ C_2, \ldots,\ C_n)) = \phi(C)\ \phi(m)(\phi(C_1),\ \phi(C_2), \ldots,\ \phi(C_n))$
- ϕ *maps collection objects of* Sig_1 *into collection objects of* Sig_2 *such that* $\phi(Collection < A_j >: X_j) = \phi(Collection < A_j >) : \phi(X_j)$.

The above definition is intended to allow a mapping of an abstract data type to its extension with possible renaming and possibly enforcing structural subtyping conditions.

The notion of schema signature morphism is extended below to a schema morphism. The extension maps the integrity constraints in accordance with the mapping of the sorts in a schema signature morphism.

Definition 12 *(***Objc** *schema morphisms) A morphism of schemas* $\phi : Sch_1 \to Sch_2$ *is a schema signature morphism* $\phi : Sig_1 \to Sig_2$ *such that* ϕ *extends to a function* $E_1 \to E_2$ *as follows:*

- $\phi(x_s) = x_{\phi(s)}$
- $\phi(a.m(a_1, a_2, \ldots, a_n)) = \phi(a).\phi(m)(\phi(a_1), \ldots, \phi(a_n))$, *where* m *may be a Boolean method.*
- $\phi(p \leftarrow p_1, p_2, \ldots, p_n) = \phi(p) \leftarrow \phi(p_1), \phi(p_2), \ldots, \phi(p_n)$

Given Definitions 10 and 11, the definition of the Sen functor is immediate and so is the verification of its functorial properties.

Definition 13 *(***Objc** *Sen functor) Define* Sen : **Sign** $\to$ **Set** *as follows:*

- $Sen(Sig)$ *is a set of sentences of the form* $\{p \leftarrow p_1, p_2, \ldots, p_n\}$ *where* $p, p_1, p_2, \ldots, p_n$ *are atoms according to Definition 10.*
- *Given a schema signature morphism* $\phi : Sig_1 \to Sig_2$,
  $Sen(\phi)(\{p \leftarrow p_1, p_2, \ldots, p_n\}) = \{\phi(p) \leftarrow \phi(p_1), \phi(p_2), \ldots, \phi(p_n)\}$.

Proposition 1 *(***Objc** *Sen functor) Then* Sen *is a functor* **Sign** $\to$ **Set**.

Proof Sen obviously preserves the identity schema signature morphism, i.e. $Sen(1_{Sig}) = 1_{Sen(Sig)}$ and the composition of schema signature morphisms, i.e., $Sen(\phi_1\phi_2) = Sen(\phi_1)Sen(\phi_2)$.

A database for a given **Objc** schema includes a domain for each sort in the schema. A domain is a set equipped with functions representing the interpretation of the method signatures. A database also includes the actual data sets, one for each collection object signature in the schema. By Definition 2, a database for a given schema must be consistent.

Definition 14 *(***Objc** *databases) A database* d *for a given schema* $Sch = (Sig, E)$ *consists of the following:*

- *A collection of sets (domains)* $\{D_s \mid s \in S\}$ *where* S *is the set of sorts of* Sig. S *includes Boolean, whose domain is true and false.*
- *For each method* m *of an interface* A_k *with the signature* $C\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n)$ *a function* $f_m : D_w \to D_C$ *where* $D_w = D_{A_k} \times D_{C_1} \times \ldots \times D_{C_n}$ *with* $w = A_kC_1 \ldots C_n$.
- *For each collection variable of type* $Collection < A_k >$ *a set with elements from the domain* D_k.
- *If* $A_l \subseteq A_k$ *then there exists a function (projection)* $D_k \to D_l$ *and thus also a function* $Collection < D_k > \to Collection < D_l >$.
- *Let* e *be a constraint with variables* X *such that* $X_s \subseteq X$ *is a set of variables in* X *of sort* s. *If* θ *is a substitution of variables in* e *(i.e. a family of functions* $X_s \to D_s$ *) then* $e < \theta >$ *evaluates to true.*

Although collections in the above definition are interpreted as sets, in general they can be bags.

To complete the above definition, we must specify its morphisms of the category of databases: a family of functions, one per database domain, that map the actual data sets as well. These maps are required to satisfy two conditions: a standard algebraic condition that applies to operations on the original domains and their images, and a condition that applies to the integrity constraints. As we are talking here about the category of databases for a schema, not just for a schema signature, database morphisms are required to map each consistent database into another consistent database.

Definition 15 *(Database morphisms for* **Objc** *schemas) Let d and d' be databases consistent with respect to a schema (Sig, E). Then a database morphism $h : D \to D'$ is a family of functions $h_s : D_s \to D'_s$ for $s \in S$ where S stands for the set of sorts of Sig.*

- *For each method m of interface A_k with signature $C\ m(C_1\ x_1, C_2\ x_2, \ldots, C_n\ x_n)$ and $a \in D_w$, $h_s(f_m(a)) = {f'}_m(h_w(a))$ holds, where h_w is a product of functions $h_{A_k} \times h_{C_1} \times \ldots \times h_{C_n}$ when $w = A_k C_1 C_2 \ldots C_n$, as in the following diagram:*

$$\begin{array}{ccc} D_w & \xrightarrow{f_m} & D_C \\ h_w \downarrow & & \downarrow h_C \\ D'_w & \xrightarrow{f'_m} & {D'}_C \end{array}$$

- *Suppose $e \in E$ has variables X. If $\theta_s : X_s \to D_s$ is a substitution of variables for X and $e < \theta >$ evaluates to true, then so does $e < \theta' >$, where θ' is constructed as the composition of θ_s and h_s.*

Definition 16 *(Db functor for* **Objc** *schemas) Define Db as follows:*

$Db(Sig)$ is a category with objects and arrows defined in 14 and 15. Given $\phi : Sig_1 \to Sig_2$, $Db(\phi)$ is defined as:

- *If d_2 is a Sig_2 database with domains $\{D_s \mid s \in S_2\}$ then $Db(\phi)(d_2)$ is a Sig_1 database with domains $\{D_s \mid s \in \phi(S_1)\}$. The same applies to the collection objects.*
- *A morphism $h : d_2 \to d'_2$ of Sig_2 databases (a family of functions $\{h_s \mid s \in S_2\}$) is mapped into a morphism $Db(\phi)(h)$ of Sig_1 databases by restricting h to the family of functions $\{h_s \mid s \in \phi(S_1)\}$.*

Proposition 2 specifies how the Db functor is constructed and how its functorial properties are verified. Note that D_s and h_s now correspond to the sort $\phi^{-1}(s) \in S_1$.

Proposition 2 *(Db functor for* **Objc** *schemas) Db in Definition 16 is a functor* **Sign** $\to$ **Cat**^{op}.

Proof A schema signature morphism $\phi : Sig_1 \to Sig_2$ according to Definition 11 maps to a database morphism $Db(\phi) : Db(Sig_2) \to Db(Sig_1)$ according to the above construction and Definition 15. The family $Db(\phi)(d_2) \to Db(\phi)(d'_2)$ is a Sig_1 database morphism, because h is a morphism of Sig_2 databases. Two properties are essential here: the injective mapping of sorts required in Definition 11 of schema signature morphisms and mapping of collections as specified in Definition 14.

The developments presented so far in this section lead to two theorems. The first one asserts that the paradigm satisfies the conditions of Definition 7 and thus is a schema transformation framework. It is based on the already established functors Sen and Db.

Theorem 2 *(A schema transformation framework)*

- *Let the category of schema signatures* **Objc** *be defined according to Definitions 8 and 11.*
- *Define the functor Sen :* **Sign** $\to$ **Set** *according to Proposition 1.*
- *Define the functor Db :* **Sign** $\to$ **Cat**^{op} *according to the Proposition 2 so that the category $Db(Sig)$ is defined according to Definitions 14 and 15.*
- *The satisfaction relation is also specified in Definitions 14 and 15.*

Under the above conditions the Integrity Requirement *holds.*

Proof We have to prove that each schema signature morphism $\phi : Sig_A \to Sig_B$ in Definition 11 satisfies the *Integrity Requirement*, that is, $d_B \models_{Sig_B} Sen(\phi)(e)$ iff $Db(\phi)(d_B) \models_{Sig_A} e$ holds for each sentence $e \in Sen(Sig_A)$.

Given $e \in Sen(Sig_A)$, $Sen(\phi)(e)$ is defined by Proposition 1. For a given object d_B of the category $Db(Sig_B)$, $Db(\phi)(d_B)$ is defined in Proposition 2. The proof amounts to:

$d_B \models_{Sig_B} (\phi(p) \leftarrow \phi(p_1), \phi(p_2), \ldots, \phi(p_n))$ iff $Db(\phi)(d_B) \models_{Sig_A} (p \leftarrow p_1, p_2, \ldots, p_n)$.

The second theorem establishes the existence of the schema-join of any two schemas of this schema transformation framework. This result provides a standard technique for integrating two **Objc** schemas. This integration is both structural and semantic (i.e. the integrity constraints are properly integrated) because it is based entirely on schema morphisms according to the pushout construction in Definition 6.

Theorem 3 *The schema-join of any two* **Objc** *schemas over the initial schema Sch_0 exists.*

Proof Given schemas Sch_1 and Sch_2 the integrated schema $Sch_1 * Sch_2$ is defined as follows: (i) $S_{Sch_1*Sch_2} = S_1 \cup S_2$ (sorts), (ii) $A_{Sch_1*Sch_2} = A_{Sch_1} \sqcup A_{Sch_2}$ (interfaces), (iii) $C_{Sch_1*Sch_2} = C_{Sch_1} \sqcup C_{Sch_2}$ (collections), and (iv) $E_{Sch_1*Sch_2} = E_{Sch_1} \sqcup E_{Sch_2}$ (constraints).

With this construction Sch_1 and Sch_2 are subschemas of $Sch_1 * Sch_2$ since we have two injective schema morphisms $\phi_1 : Sch_1 \to Sch_1 * Sch_2$ and $\phi_2 : Sch_2 \to Sch_1 * Sch_2$.

Suppose that we are given $p : Sch_1 \to S_{12}$ and $q : Sch_2 \to S_{12}$. The required schema morphism $\phi : Sch_1 * Sch_2 \to Sch_{12}$ is defined on the sorts as follows: $\phi(s) = p(s)$ for $s \in S_1$ and $\phi(s) = q(s)$ for $s \in S_2$.

Note that $\phi_1(s) = s$ for $s \in S_0$ and $\phi_2(s) = s$ for $s \in S_0$. So ϕ is well defined for $s \in S_1 \cap S_2$.

As interfaces, collections and constraints of $Sch_1 * Sch_2$ are defined as disjoint unions of the corresponding components of Sch_1 and Sch_2, we have $\phi h = p$ and $\phi k = q$.

Note that with the choice of a different logic the above result would not necessarily hold. The union of constraints could lead to a set of sentences for which there is no database (particularly in a given category) that satisfies the constraints (represents a model for the constraints) $\phi_1(E_1)$ and $\phi_2(E_2)$ (see Definition 3).

7 Conclusions and Related Work

The main contribution of this paper is a model theory for generic schema management. While the ideas on generic schema management proposed so far have been mostly informal [4] and pragmatic [5], this paper shows that a formal framework for such generic tools does indeed exist.

A distinctive feature of the presented paradigm is that it applies to a variety of categories of schemas and to a variety of logic bases for expressing database integrity constraints. This level of generality is accomplished by making use of a categorical model theory called *institutions*, proposed for programming language paradigms [9]. We believe ours is the first application of this theory to database models and languages. An earlier attempt in [15] is similar in spirit, but is less general. It also does not address the logic basis and preservation of integrity constraints and lacks provable results.

The core of this model theory is a general, transformation-oriented definition of a data model. This generic schema management framework serves as a pattern for constructing data models in such a way that schema transformations as well as the associated database transformations have well-defined meaning. As a rule, most well known data models have not been constructed in such a spirit. Furthermore, this framework is intended to be applied to data models that are still not completely or formally established, such as XML. A further distinctive feature of this framework is that it has a strong database integrity requirement as its possibly most fundamental component.

This paper also shows how to address some well-known problems such as schema equivalence [13, 14] and schema integration [6] at this level of generality. The results are independent of a particular category of schemas and apply to both structural properties and integrity constraints. The requirement for proper handling of the integrity constraints in those problems is one of the contributions of this paper. Furthermore, this approach is independent of a particular logic paradigm used as a basis for the constraint language.

Contrary to the published work on schema and data transformations (schema integration in particular) in which transformations at the meta and data levels have the same direction [19], this paper reveals a subtlety not considered in the relevant papers. The subtlety is manifested in the reversal of the transformation arrows at the two levels. This observation is an important component of the presented model theory, having both pragmatic and mathematical significance.

In this paper one option is that schema transformations must satisfy particular structural subtyping conditions. Such a strict discipline has its place in programming languages, but it is often not satisfied in schema transformations. Furthermore, if the abstract data types are equipped with constraints, semantic compatibility notions such as behavioral subtyping [16] become a major issue. Model theoretic implications of behavioral compatibility issues in mapping abstract data types equipped with constraints are given in [1].

In a model in which inheritance is identified with subtyping (as in Java) a single partial order of sorts is required. This introduces further subtleties in schema transformations and schema integration. Models with different types of ordering of sorts are elaborated in [1] and [3]. The relevant results on pushouts of algebraic specifications are given in [12].

In this paper we do not consider the problem of mapping one schema transformation framework (for example, relational) into another (for example, object-oriented). This situation is captured by the notion of a morphism of schema transformation frameworks (following [9, 15]), and is a topic of work in progress.

The formal paradigm presented in this paper applies to appropriately simplified XML schemas [23]. The reasons are: the nature of the type system of the XML Schema, the kind of the integrity constraints (keys and referential integrity constraints) expressible in XML schemas (also considered in [8]), and the existence of the initial schema (more precisely, a name space). Space limitations do not allow further elaboration of these important implications of the development presented in this paper. However, application of this model theory to the XML data model is a major topic of future research.

This paper is not addressing explicitly modeling of database dynamics. But this is by no means a limitation of the paradigm. Database dynamics is taken into account by choosing the Sen functor for a suitable temporal logic. A development of such a paradigm is given in [1, 2, 3].

Acknowledgments

We are grateful for many helpful suggested improvements from Alon Halevy, Svetlana Kouznetsova, Renée Miller, and the anonymous referees.

References

[1] Alagić, S.: Semantics of temporal classes, *Information and Computation, 163*, pp. 60 - 102, 2000.

[2] Alagić, S.: Constrained matching is type safe, Proceedings of the Sixth Int. Workshop on Database Programming Languages, *LNCS 1369*, Springer-Verlag, pp. 78-96, 1998.

[3] Alagić, S. and M. Alagić, Order-sorted model theory for temporal executable specifications, *Theoretical Computer Science 179*, pp. 273-299, 1997.

[4] Bernstein, P. A. , A. Halevy, R. A. Pottinger: A vision for management of complex models, *ACM SIGMOD Record 29(4)*, 2000.

[5] Bernstein, P.A. and E. Rahm: Data warehouse scenarios for model management. ER 2000, *LNCS 1920*, Springer-Verlag, pp. 1-15, 2000.

[6] Batini, C., M. Lenzerini, and S. B. Navathe: A comparative analysis of methodologies for database schema integration, *ACM Computing Surveys* 18(4), pp. 323-364, 1986.

[7] Buneman, P., S. Davidson, and A. Kosky: Theoretical aspects of schema merging. EDBT 1992, pp. 152-167.

[8] Fan, W. and L. Libkin, On XML integrity constraints in the presence of DTDs, *Proc. of ACM PODS Conf.*, 2001.

[9] Goguen, J. and R. Burstall, Institutions: Abstract model theory for specification and programming, *Journal of the ACM, 39*, No. 1, pp. 92-146, 1992.

[10] Goguen, J.: Types as theories, in: G. M. Reed, A. W. Roscoe and R. F. Wachter, *Topology and Category Theory in Computer Science*, pp. 357-390, Clarendon Press, Oxford, 1991.

[11] Goguen, J. and J. Meseguer, Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations, *Theoretical Computer Science 105*, pp. 217-273, 1992.

[12] Haxthausen, A.E. and F. Nickl: Pushouts of order-sorted algebraic specifications, in M. Wirsing and M. Nivat, eds., Algebraic Methodology and Software Tech. (AMAST 96), *LNCS 1101*, Springer-Verlag, pp. 132-147, 1996.

[13] Hull, R.: Relative information capacity of simple relational database schemata, *SIAM Journal of Computing, Vol. 15*, No. 3, pp.856 -886, 1986.

[14] Hull, R.: Managing semantic heterogeneity in databases: A theoretical perspective, *PODS 1997*, 51-61

[15] Kalinichenko, L.A.: Methods and tools for equivalent data model mapping construction, *Proc. of EDBT, LNCS 416*, Springer-Verlag, pp. 92-119, 1990.

[16] Liskov, B. and J. M. Wing: A behavioral notion of subtyping, *ACM TOPLAS* **16**, pp. 1811 1841.

[17] Mac Lane, S.: *Categories for a Working Mathematician*, Springer, 1998.

[18] Madhavan, J., P. A. Bernstein, E. Rahm: Generic schema matching with Cupid, *VLDB 01*, to appear.

[19] Miller, R. J., Y. E. Ioannidis, R. Ramakrishnan, Schema equivalence in heterogeneous systems: Bridging theory and practice, *Information Systems Vol. 19*, No. 1, pp. 3-31, 1994.

[20] Rahm, E. and P.A. Bernstein: On matching schemas automatically. MSR Tech. Report MSR-TR-2001-17, 2001.

[21] Spaccapietra, S., C. Parent, Y. Dupont: Model independent assertions for integration of heterogeneous schemas, *VLDB Journal 1*, pp. 81-126, 1992.

[22] Spaccapietra, S. and C. Parent: View integration: A step forward in solving structural conflicts, *IEEE TKDE 6(2)*, pp. 258 - 274, 1994.

[23] W3C: XML Schema, http://www.w3c.org/XML/schema, 2001.